САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Кафедра математической теории игр и статистических решений

Смирнов Дмитрий Сергеевич

Магистерская диссертация

Одна модификация задачи о многоруком бандите

Направление 01.04.02

Прикладная математика и информатика

Магистерская программа «Исследование операций и системный анализ»

Научный руководитель,

кандидат физ.-мат. наук,

доцент

Громова Е. В.

Санкт-Петербург

2017

Содержание

1. Введение

Машинное обучение (Machine Learning) — чрезвычайно широкая и активно развивающаяся область искусственного интеллекта, которая находится на стыке математической статистики, методов оптимизации и фундаментальных математических дисциплин. Методы машинного обучения все глубже проникают в самые разные области человеческой деятельности.

Одним из разделов машинного обучения является «обучение с подкреплением» [29] (reinforcement learning), в котором ключевой задачей выступает так называемая дилемма «исследования-использования». Задача исследования-использования заключается в поиске компромисса между обучением и применением ранее полученных знаний с целью максимизации выигрыша. Такие задачи часто возникают в реальных ситуациях, например, в клинических исследованиях, динамической (адаптивной) маршрутизации, в финансах и интернет-маркетинге.

Одним из способов формализации дилеммы исследования-использования является стохастическая *задача о многоруком бандите* (multi-armed bandit problem), так же известная как задача оптимального управления в случайной среде [2]. Формальное определение задачи мы дадим ниже, а здесь ограничимся лишь её словесным описанием. Представим ситуацию, в которой игрок многократно выбирает одну из n альтернатив (действий), каждый раз получая за это некоторую награду, величина которой зависит от неизвестного вероятностного распределения выбранной альтернативы. С этого момента каждый такой выбор будем называть *игрой*. Задача игрока состоит в том, чтобы максимизировать суммарный выигрыш за конечное число последовательных игр. Заметим, что в качестве игрока, как правило, выступает некоторая автоматизированная система. Каждую альтернативу можно интерпретировать как рычаг игрового

автомата, называемого «однорукий бандитом». Тогда задача представляется в виде игрового автомата с несколькими рычагами, который по аналогии можно назвать «многоруким бандитом».

Как было отмечено, задача о многоруком бандите находит свое применение в различных сферах человеческой деятельности [6, 13, 16, 17, 27]. Отдельно стоит отметить её многочисленные приложения в интернет-маркетинге [5, 21, 23, 25].

Одной из ключевых задач интернет-маркетинга является увеличение *конверсии* сайта. Тестирование интернет-страниц — один из способов решения данной задачи. Под конверсией сайта понимается отношение количества посетителей сайта, выполнивших необходимое действие (покупка товара, заказ обратного звонка, подписка на рассылку и т. д.), к общему числу посетителей. Тестирование интернет-страниц, как будет показано ниже, может быть сведено к задаче о многоруком бандите. Такой подход в сравнении с классическим позволяет увеличить конверсию за время тестирования.

На практике в условиях задачи о многоруком бандите кроме значений полученных выигрышей от выбора альтернатив часто доступна некоторая дополнительная информация, которую можно использовать для улучшения выбора. В интернет-маркетинге это может быть индивидуальная информация о посетителе веб-ресурса, например, источник перехода, предполагаемый пол, возраст и т. д. Модификация задачи о многоруком бандите, которая учитывает дополнительную информацию, называется *задачей о контекстном бандите.* В данной работе будет показано, как, используя дополнительную информацию об источнике перехода, увеличить конверсию за время тестирования.

В работе также выделяется особый тип дополнительной информации, содержащий прямые рекомендации по выбору альтернатив. Такого рода информация формализуется в виде *экспертных подсказок*. Для учета подсказок

эксперта модифицирован алгоритм, решающий задачу о многоруком бандите.

Актуальность исследования данной темы обусловлена возрастающей значимостью интернет-маркетинга для бизнеса и экономики в совокупности с бурным ростом числа приложении методов машинного обучения в данной сфере.

Новизна работы состоит, во-первых, в предложенном подходе к тестированию интернет-страниц на основе задачи о контекстном бандите, во-вторых, в модификации алгоритма UCB1 для решения задачи о многоруком бандите с экспертами.

Основные результаты выпускной квалификационной работы опубликованы в работах [6, 5] и докладывались на двух конференциях: XLVII и XLVIII международной конференции «Процессы управления и устойчивость» Control Processes and Stability (CPS).

Работа состоит из введения, трех глав и заключения.

В главе 1 дается классическая формулировка стохастической задачи о многоруком бандите. Приводится обзор алгоритмов, используемых для решения данной задачи. Тестирование интернет-страниц с целью увеличения конверсии сводится к задаче о многоруком бандите. Демонстрируются результаты численного эксперимента на основе программной реализации симуляции тестирования.

В главе 2 продолжается тема тестирования интернет-страниц. Здесь предложен новый подход к тестированию, основанный на задаче о контекстном бандите. Дается подробное описание алгоритма LinUCB. Разработана программная реализация симуляции тестирования. Приведены результаты численных экспериментов

В главе 3 формулируется задача о многоруком бандите с экспертами. Предложена модификация алгоритма UCB1. Продемонстрированы результаты

численных экспериментов.

1.1. Обзор литературы

Первая глава работы посвящена задаче о многоруком бандите, которая впервые рассмотрена Гербертом Роббинсом (Herbert Ellis Robbins) в 1952 в статье [24]. В другой работе [19] того же автора вводится так называемая *функция сожаления* и доказывается, что она асимптотически не меньше $ln(T)$, или более формально, $R(T) = \Omega(ln(T))$. Здесь же описывается семейство алгоритмов UCB (upper confidence bound).

В работе [29] представлены некоторые алгоритмы для решения задачи о многоруком бандите. Среди них ε-жадный (ε-greedy), softmax, алгоритм преследования (pursuit) и алгоритм сравнения с подкреплением (reinforcement comparison). В книге приводятся результаты численных экспериментов, однако отсутствует теоретический анализ функций сожаления описанных алгоритмов.

Большую значимость имеет работа [10], в которой предложены алгоритм UCB1 и модифицированный ε-жадный алгоритм (ε_n-greedy). Кроме того, там же рассматриваются модификации алгоритма UCB1: UCB1-Normal (для нормального распределения) и UCB1-Tuned (учитывает дисперсию). Для всех алгоритмов, за исключением последнего, найдены теоретические оценки функции сожаления.

Описание алгоритма Thompson Sampling содержится в работе [8]. Однако идеи, лежащие в его основе, предложены гораздо ранее [30]. Алгоритм play-the-winner описан в статье [26].

Во второй главе рассматривается так называемая *задача о контекстном бандите* (contextual bandit problem). В литературе данную задачу можно встретить под разными названиями, включая следующие: «задача об ассоциативном

бандите» (associative bandit problem) [28], «задача о многоруком бандите с советами эксперта» (multi-armed bandit problem with expert advice) [11], «задача о многоруком бандите с внешней информацией» (multi-armed bandit problem with side information) [22] и «задача о многоруком бандите с ковариатами» (bandit problems with covariates) [32]. Название «contextual bandit problem» впервые использовано в статье [20], причем настолько удачно, что в дальнейшем стало основным.

В работе рассматривается задача о контекстном бандите с линейными функциями выигрышей. Данный частный случай [12] широко представлен в литературе и имеет множество практических приложений. Например, в работе [7] авторы используют линейные функции для прогнозирования вероятности перехода по рекламному объявлению, а в статье [21] — для прогнозирования заинтересованности пользователей статьями. Мы же, в свою очередь, используем данный подход для оценки вероятности достижения конверсионной цели.

Литература, посвященная исследуемой теме, изобилует алгоритмами для решения задачи о контекстном бандите. Наиболее широко представлены в литературе LinUCB [21], Epoch-greedy [20], LinRel [12], Thompson Sampling [9].

В приложении задачи о контекстном бандите для тестирования интернет-страниц мы используем алгоритм LinUCB. Данный алгоритм впервые рассмотрен в работе [21], где его эффективность была продемонстрирована экспериментально. В статье [15] проводится теоретический анализ функции сожаления алгоритма LinUCB, результатом которого являются её нижняя и верхняя оценки.

Наконец, отметим работы, посвященные многочисленным приложениям задачи о многоруком (контекстном) бандите: клинические исследования (назначение методов лечения, минимизируя потери среди пациен-

тов) [17, 16], адаптивная маршрутизация (минимизация задержек в сети) [13], финансы (формирование инвестиционного портфеля) [27], тестирование интернет-страниц с целью увеличения конверсии [6, 5], персонализация web-контента [21], динамическое изменение цен [25], публикация контента в социальных сетях [18] и интернет-реклама (выбор рекламных объявлений для максимизации дохода) [23].

2. Глава 1. Тестирование интернет-страниц на основе задачи о многоруком бандите

В настоящей главе рассматривается приложение задачи о многоруком бандите для тестирования интернет-страниц. В начале главы формулируются задачи. Во втором параграфе главы дается формальное определение стохастической задачи о многоруком бандите. Вводится понятие функции сожаления и стратегии. В третьем параграфе представлен обзор алгоритмов, используемых для решения задачи о многоруком бандите. Четвертый параграф главы посвящен непосредственно тестированию интернет-страниц, которое формализуется в виде задачи о многоруком бандите. Пятый параграф заключается в численном эксперименте, который состоит в программной симуляции тестирования. Там же продемонстрированы результаты работы представленных в обзоре алгоритмов. В конце главы сформулированы выводы.

2.1. Постановка задачи

Тестирование интернет-страниц — один из инструментов увеличения конверсии сайта. Для проведения тестирования могут быть применены алгоритмы для решения задачи о многоруком бандите. Задачи текущей главы: провести обзор известных алгоритмов, разработать программную реализацию симуляции тестирования, продемонстрировать и проанализировать результаты применения алгоритмов.

2.2. Задача о многоруком бандите

Перейдем к формальному описанию стохастической задачи о многоруком бандите [17]. Пусть имеется набор из n действий $a_1, \ldots, a_n$. Каждому дей-

ствию a_i, $i = 1, \ldots, n$ соответствует случайная величина $\xi_i \sim D_i$ с математическим ожиданием μ_i и дисперсией σ_i^2. Вид распределения D_i, математическое ожидание μ_i и дисперсия σ_i^2 игроку неизвестны. В каждый момент времени $t = 1, \ldots T$ игрок выбирает действие $a_{j(t)}$ и получает выигрыш $r_{j(t)}$, который является реализацией соответствующей случайной величины $\xi_{j(t)} \sim D_{j(t)}$. Игрок преследует две цели: во-первых, определить действие, дающее наибольший средний выигрыш, и во-вторых, получить наибольший суммарный выигрыш за время T.

Для решения описанной задачи существуют алгоритмы (стратегии), которые в каждый момент времени t определяют, какое действие следует выбрать. Распространенной мерой производительности данных алгоритмов является *функция суммарного сожаления*, определяемая следующим образом:

$$R(T) = \mu^* T - \sum_{t=1}^{T} \mu_{j(t)},$$

где $\mu^* = \max_{1 \leqslant i \leqslant n} \mu_i$.

Если для функции сожаления справедлива оценка $R(T) = O(\ln(T))$, то говорят, что алгоритм решает задачу о многоруком бандите. Иногда также такой алгоритм называют *оптимальным*.

В дальнейшем будем рассматривать задачу о многоруком бандите, в которой случайные величины $\xi_1, \ldots, \xi_n$ имеют распределение Бернулли [1] с неизвестными параметрами $p_1, \ldots, p_n$. Таким образом, выигрыш игрока в результате выбора действия равен 0 или 1.

Иногда мы будем использовать словосочетание *лучшее (оптимальное) действие*, понимая под этим действие, которому соответствует случайная величина с наибольшим математическим ожиданием. Слова *алгоритм* и *стратегия* будем использовать как синонимы.

2.3. Обзор алгоритмов

2.3.1. Алгоритм play-the-winner

Простейший эвристический алгоритм play-the-winner [26] заключается в многократном выборе одного и того же действия до первого проигрыша. Следующее действие может выбираться как случайно, так и поочередно. Понятно, что в случае, когда математическое ожидание лучшего действия близко к единице, рассматриваемый алгоритм будет давать суммарный выигрыш, близкий к выигрышу оптимального алгоритма. В остальных случаях данная эвристика уступает алгоритмам, рассмотренным ниже.

2.3.2. ε-жадный алгоритм (ε-greedy)

Благодаря простоте реализации алгоритм ε-greedy [29] получил широкое применение в оптимизационных задачах в условиях неопределенности. Алгоритм работает следующим образом: в каждый момент времени $t = 1, \ldots, T$ с вероятностью $1 - \varepsilon$ выбирается действие с наибольшим средним выигрышем и с вероятностью ε — случайное действие.

Формально, пусть после t игр средние выигрыши действий $a_1, \ldots, a_n$ равны соответственно $\overline{p}_1(t), \ldots, \overline{p}_n(t)$, тогда вероятность выбора действия a_i в следующий момент времени вычисляется по формуле:

$$\widetilde{p}_i(t+1) = \begin{cases} 1 - \varepsilon + \frac{\varepsilon}{n}, & \text{если } i = \arg\max_{j=1,\ldots,n} \overline{p}_j(t); \\ \frac{\varepsilon}{n}, & \text{в противном случае.} \end{cases}$$

Особенностью алгоритма является четкое разделение на две фазы: с вероятностью $1 - \varepsilon$ алгоритм находится в фазе использования, в которой выбирает оптимальное на текущий момент действие, и с вероятностью ε — в фазе исследования, в которой происходит обучение алгоритма.

Несмотря на свое название алгоритм ε-greedy нельзя отнести к так называемым *жадным алгоритмам*, поскольку в фазе исследования данная стратегия может выбирать неоптимальное на текущий момент действие. Таким образом, «степень жадности» алгоритма зависит от значения параметра ε. Очевидно, при $\varepsilon = 0$ алгоритм становится жадным, а при $\varepsilon = 1$ осуществляет выбор случайным образом.

Одним из недостатков описанного алгоритма является необходимость в подборе параметра ε. Другой недостаток вытекает из следующих соображений. Вероятность попадания в фазу исследования всегда постоянна и равна ε, однако на более поздних этапах работы алгоритма нецелесообразно тратить столько же времени на обучение алгоритма, сколько и в начале его работы. Эта идея реализована в алгоритме ε_n-greedy.

Заметим, что настоящий алгоритм не является оптимальным, поскольку функция сожаления имеет линейный рост.

2.3.3. Модифицированный ε-жадный алгоритм (ε_n-greedy)

Алгоритм ε_n-greedy [10] осуществляет выбор действий подобно предыдущему алгоритму, однако перед каждым выбором вероятность ε пересчитывается по формуле:

$$\varepsilon_t = \min\left(1, \frac{cn}{d^2 t}\right),$$

где n — число действий, c и d — параметры, причем $0 < d < \min \Delta_i$, $\Delta_i = p^* - p_i, p^* = \max_{1 \leqslant i \leqslant n} p_i, c \geqslant 0$.

В [10] доказана оптимальность данного алгоритма.

Отметим недостатки, присущие двум приведенным выше алгоритмам. Первый недостаток заключается в том, что действия в фазе исследования вы-

бираются случайным образом, независимо от средних выигрышей, что, конечно, приводит к снижению суммарного выигрыша.

Другим недостатком является то, что вероятность попадания в фазу исследования никак не зависит от величины полученных выигрышей, в то время как при наличии нескольких действий с близкими средними выигрышами эту вероятность следует увеличить.

2.3.4. Алгоритм softmax

Идея алгоритма softmax [29] состоит в том, чтобы производить выбор действий согласно вероятностям $\widetilde{p}_1, \ldots, \widetilde{p}_n$, пропорциональным средним выигрышам. Для вычисления вероятностей воспользуемся распределением Гиббса, однако отметим, что это можно сделать и другими способами.

Пусть после t игр средние выигрыши действий равны соответственно $\overline{p}_1(t), \ldots, \overline{p}_n(t)$, тогда вероятность того, что в момент времени $t+1$ будет выбрано действие a_i равна

$$\widetilde{p}_i(t+1) = \frac{e^{\overline{p}_i(t)/\tau}}{\sum_{j=1}^{n} e^{\overline{p}_j(t)/\tau}}, i = 1, \ldots, n,$$

где τ — положительный параметр, называемый *температурным коэффициентом*. При $\tau \to +\infty$ действия выбираются случайным образом, при $\tau \to 0$ алгоритм становится жадным.

Из недостатков стоит выделить наличие параметра τ, который необходимо подбирать, а также отсутствие теоретического анализа функции сожаления.

В работе [14] предложена вариация алгоритма, в которой параметр τ уменьшается со временем. Там же доказано, что данная вариация softmax является оптимальной.

2.3.5. Алгоритм UCB1

Все приведенные выше алгоритмы при выборе действия основывались только на значениях средних выигрышей. Алгоритм UCB1 [10] учитывает и то, сколько раз выбиралось каждое действие.

Семейство алгоритмов UCB (Upper Confidence Bounds) описано в [19]. Рассмотрим самый простой в реализации и популярный алгоритм UCB1. На каждом шаге алгоритм выбирает действие с индексом:

$$j(t) = \arg\max_{i=1,\dots,n} \left(\overline{p}_i + \sqrt{\tfrac{2\ln t}{n_i}} \right), \tag{1}$$

где n_i — количество раз, когда выбиралось действие a_i.

Очевидными преимуществами алгоритма являются отсутствие параметров и детерминированность. В [10] показано, что для функции сожаления $R(T)$ данного алгоритма справедлива оценка:

$$R(T) \leqslant 8 \sum_{i:p_i<p^*} \left(\tfrac{\ln T}{\Delta_i} \right) + \left(1 + \tfrac{\pi^2}{3} \right) \Big(\sum_{j=1}^{n} \Delta_j \Big),$$

где как и ранее $\Delta_i = p^* - p_i$. Таким образом, UCB1 — оптимальный алгоритм.

Существуют различные вариации алгоритма UCB1. Например, в работе [10] предложен алгоритм UCB1-Normal для случая, когда случайные величины, соответствующие действиям $a_1, \dots, a_n$, распределены нормально.

В той же статье предлагается модификация алгоритма UCB1, названная UCB1-Tuned, которая, по мнению авторов, на практике может работать лучше, чем UCB1, однако не имеет никаких теоретических гарантий. Главной особенностью алгоритма UCB1-Tuned является учет выборочной дисперсии для выбора следующего действия.

2.3.6. Метод преследования (pursuit algorithm)

Еще одним классом алгоритмов являются так называемые методы преследования (pursuit methods). Рассмотрим алгоритм, который представлен в [29].

В каждый момент времени $t = 1, \ldots, T$ алгоритм вычисляет вероятности $\widetilde{p}_1, \ldots, \widetilde{p}_n$ выбора действий $a_1, \ldots, a_n$ по следующей формуле:

$$\widetilde{p}_i(t+1) = \begin{cases} \widetilde{p}_i(t) + \beta(1 - \widetilde{p}_i(t)), & \text{если } i = \arg\max_{j=1,\ldots,n} \overline{p}_j(t); \\ \widetilde{p}_i(t) + \beta(0 - \widetilde{p}_i(t)), & \text{в противном случае.} \end{cases}$$

На первом шаге считаем, что $\widetilde{p}_i(0) = \frac{1}{n}$. Параметр β называется *скоростью обучения*.

Таким образом, вероятность выбора действия с наибольшим средним выигрышем всегда увеличивается, как бы «преследуя» это действие.

Недостатками алгоритма являются наличие параметра β, который необходимо подбирать, а также отсутствие теоретического анализа функции сожаления.

2.3.7. Метод сравнения с подкреплением (Reinforcement Comparison)

Ключевой идеей всех алгоритмов, используемых в обучении с подкреплением, является организация выбора действий таким образом, чтобы действия с большими выигрышами выбирались чаще, а с меньшими — реже. Но как определить, какие действия имеют «большой» выигрыш, а какие — «маленький»? Можно, например, сравнивать выигрыши действий с некоторой эталонной (предпочитаемой) величиной выигрыша: если выигрыш больше этой величины, то он считается «большим», если меньше — «маленьким». Методы обучения, основанные на таком подходе, называются *методами сравнения*

с подкреплением (reinforcement comparison). Рассмотрим алгоритм, который представлен в [29].

В каждый момент времени $t = 1, \ldots, T$ алгоритм вычисляет вероятности $\widetilde{p}_1, \ldots, \widetilde{p}_n$ выбора действий $a_1, \ldots, a_n$ по следующей формуле:

$$\widetilde{p}_i(t) = \frac{e^{\pi_i(t)}}{\sum_{j=1}^{n} e^{\pi_j(t)}},$$

где $\pi(t) = (\pi_1(t), \ldots, \pi_n(t))$ — вектор *предпочтений* действий $a_1, \ldots, a_n$.

Если выбрано действие $a_{j(t)}$ и получен выигрыш $r(t)$, то предпочтение $\pi_{j(t)}$ пересчитывается по следующей формуле:

$$\pi_{j(t)}(t+1) = \pi_{j(t)}(t) + \beta(r(t) - \overline{r}(t)),$$

где $\beta \in (0, 1)$ — параметр алгоритма, $\overline{r}(t)$ — предпочитаемый (эталонный) выигрыш, который каждый раз пересчитывается по формуле:

$$\overline{r}(t+1) = (1 - \alpha)\overline{r}(t) + \alpha r(t),$$

где $\alpha \in (0, 1)$ — параметр алгоритма.

При инициализации алгоритма начальные предпочтения могут быть установлены равными нулю. Начальный предпочитаемый выигрыш следует подбирать исходя из априорных знаний о выигрышах, если они имеются, либо исходя из других соображений.

Недостаток алгоритма — наличие параметров α и β. Теоретическая оценка функции сожаления данного алгоритма в литературе не найдена.

2.3.8. Алгоритм Thompson Sampling

Thompson Sampling один из самых старых алгоритмов для решения задач исследования-использования. Идеи, на которых основан алгоритм, появились задолго до возникновения задачи о многоруком бандите [30].

В основе алгоритма лежит факт из байесовской статистики: для случайной величины, имеющей распределение Бернулли с параметром p, в качестве сопряженного априорного распределения выступает бета-распределение с плотностью

$$f_p(x) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{\widehat{B}(\alpha,\beta)},$$

где $\widehat{B}(\alpha,\beta)$ — бета-функция. Другими словами, параметр p распределения Бернулли имеет бета-распределение с параметрами α и β.

Зная это, можно показать, что в случае, когда параметр p имеет априорное бета-распределение с параметрами α и β, если после n игр было s выигрышей и f неудач, то апостериорное распределение параметра p есть $B(s+\alpha, f+\beta)$. На этом основана работа алгоритма.

Изначально считаем, что параметр $p_i, i = 1,\ldots,n$ имеет равномерное распределение на отрезке $[0,1]$. Известно, что непрерывное равномерное распределение есть частный случай бета-распределения, т. е. $U[0,1] = B(1,1)$, где $B(\alpha,\beta)$ — бета-распределение с параметрами α,β.

Опишем подробно первый шаг алгоритма. На первом шаге генерируются параметры $p_1,\ldots,p_n$ из непрерывного равномерного распределения на отрезке [0, 1], т. е. из $B(1,1)$. Выбирается действие с наибольшим сгенерированным параметром. Пусть это действие $a_{j(t)}$. Далее, если выигрыш равен 1, то параметр $p_{j(t)}$ имеет распределение $B(2,1)$, если проигрыш, то $B(1,2)$. Далее параметр $p_{j(t)}$ будет генерироваться из бета-распределения с обновленными параметрами α и β.

Далее каждый раз алгоритм генерирует параметры $p_1,\ldots,p_n$, выбирает действие с наибольшим параметром, в зависимости от выигрыша обновляет параметры бета-распределения и повторяет действие.

В работе [8] доказывается оптимальность алгоритма.

2.4. Тестирование интернет-страниц

В наши дни практически каждая компания имеет сайт в сети интернет, с помощью которого можно купить товар или заказать услугу. Для увеличения числа клиентов в сети интернет необходимо решить две задачи. Во-первых, повысить посещаемость сайта, во-вторых, увеличить его *конверсию*. Под *конверсией* сайта понимается отношение количества посетителей сайта, выполнивших необходимое действие (покупка товара, заказ обратного звонка, подписка на рассылку и т. д.), к общему числу посетителей. На практике, как правило, вместо конверсии сайта отслеживают конверсию отдельных страниц или конверсию определенных действий.

Для решения первой задачи можно воспользоваться, например, контекстной рекламой [4]. Мы же рассмотрим здесь один из способов решения второй задачи. Одним из инструментов увеличения конверсии является так называемое *A/B-тестирование*. Для понимания дальнейшего изложения важно отметить, что на конверсию сайта влияют его дизайн, расположение элементов, шрифты, содержание и т. д. Классический вариант тестирования проводится следующим образом. Имеются n (часто $n = 2$) версий одной страницы, которые отличаются, как правило, одним элементом, например, цветом кнопки «купить». На этих страницах устанавливается определенная цель, например, клик по кнопке «купить». Показатели достижения цели фиксируются. Все посетители сайта случайным образом делятся на n групп, каждой из которых показывается своя версия страницы. При достижении статистически значимого числа показов, сравниваются числовые показатели достижения цели и определяется вариант страницы с наибольшей конверсией.

Данный способ проведения тестирования содержит один существенный недостаток. При тестировании описанным способом в равных пропорциях по-

казываются страницы с низкой и высокой конверсией. Поскольку показы страниц с низкой конверсией несут убытки компании, то целесообразно во время тестирования минимизировать число показов таких страниц. Иначе говоря, для компании, проводящей тестирование страниц, важно не только выявить версию страницы с наибольшей конверсией, но и во время тестирования достичь наибольшую суммарную конверсию. Последнего можно добиться с помощью алгоритмов для решения задачи о многоруком бандите.

Формализуем тестирование интернет-страниц в виде задачи о многоруком бандите. Пусть набор действий $a_1, \ldots, a_n$ соответствует тестируемым страницам. Выбор действия a_i означает показ страницы с номером i. Достижение конверсионной цели при показе страницы соответствует получению выигрыша, равного 1. Если при показе страницы цель не достигнута, выигрыш равен 0. Таким образом, тестирование страниц свелось к задаче о многоруком бандите, в которой случайные величины, соответствующие действиям, имеют распределение Бернулли с математическим ожиданием, равным конверсиям страниц.

2.5. Численный эксперимент

В рамках данной работы не ставилась задача провести тестирование реальных веб-страниц. Цель данной главы — показать каким образом задача о многоруком бандите может быть применена для тестирования страниц, провести обзор известных алгоритмов, а также продемонстрировать результаты их работы. Для достижения последнего была выполнена программная реализация симуляции тестирования на языке Python. В силу громоздкости программы в приложении 1 приведены лишь программные коды алгоритмов. Полностью же программа доступна в [3].

Программа принимает на вход количество тестируемых страниц n, вектор конверсий страниц $(p_1, \ldots, p_n)$ и количество показов T. При выборе страницы (действия) с номером i программа генерирует выигрыш как реализацию случайной величины, имеющей распределение Бернулли с параметром конверсии p_i (i-ая компонента вектора конверсий).

В качестве результатов работы программы выводится величина конверсии, достигнутой каждым алгоритмом, номер страницы с наибольшей конверсией, число показов каждой страницы, а также графики функции сожаления и достигаемой конверсии. Под конверсией в данной случае понимается отношение числа достигнутых целей к общему числу показов страницы.

Важно отметить, что каждый запуск алгоритмов на одних и тех же данных дает разный результат. Такое поведение алгоритмов обусловлено двумя причинами: во-первых, задача о многоруком бандите является стохастической, во-вторых, некоторые алгоритмы используют случайность. Поэтому, для того чтобы облегчить интерпретацию результатов, делается 100 запусков программы и анализируются средние значения выходных данных.

Ряд алгоритмов (ε-greedy, ε_n-greedy, softmax, pursuit, reinforcement comparison) зависит от параметров, которые напрямую влияют на их производительность. Это можно видеть на рис. 1, на котором показана конверсия, достигаемая в процессе тестирования страниц при использовании алгоритма ε-greedy с разными значениями параметра ε. Таким образом, перед запуском алгоритмов необходимо осуществить подбор их параметров. При выборе числового значения параметра можно отталкиваться от его содержательного значения в алгоритме. Например, параметр ε в алгоритме ε-greedy отвечает за вероятность попадания в фазу исследования. Уменьшение значения параметра делает алгоритм жадным. Однако не во всех алгоритмах смысл параметра лежит на поверхности. Примерами таких алгоритмов являются softmax и

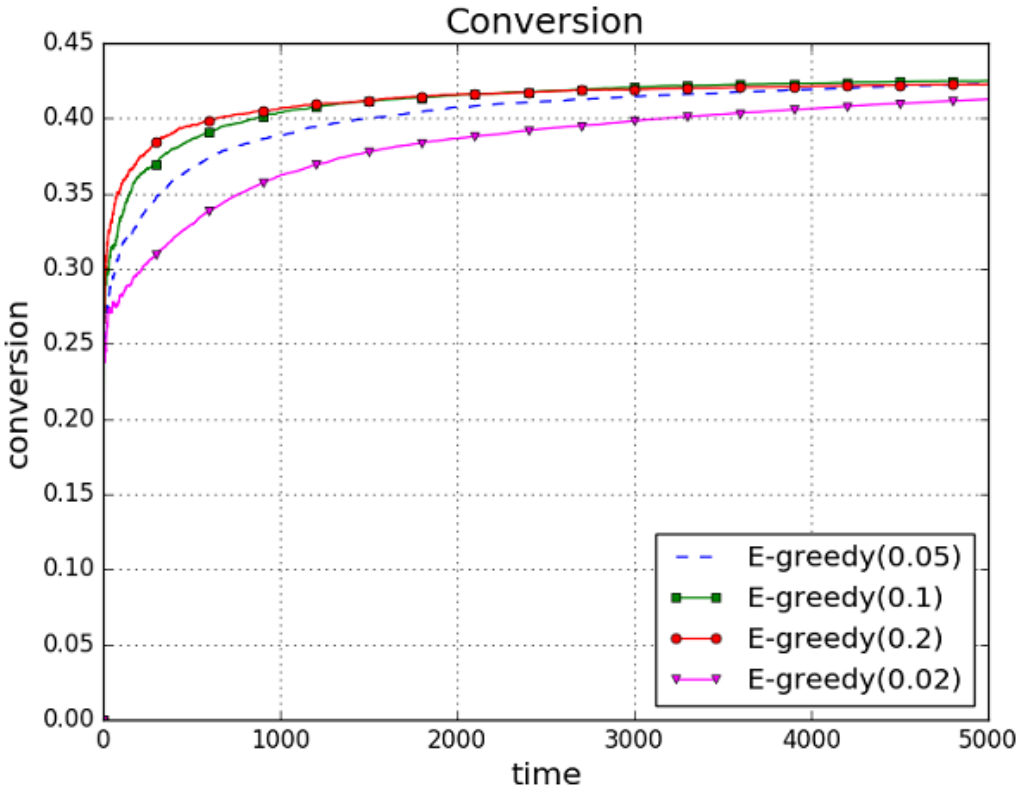

Рис. 1: Конверсия в ходе тестирования (ε-greedy с разными параметрами)

reinforcement comparison.

Также заметим, что алгоритм с подобранным значением параметра, дающий хороший результат на одних входных данных, может иметь неудовлетворительный результат на других данных. Таким образом, при подборе параметров необходимо учитывать входные данные.

В реальных ситуациях подбор параметров следует осуществлять путем предварительных прогонов алгоритма на тестовых данных, например, с помощью описанной выше симуляции.

Рассмотрим сначала случай тестирования двух страниц и продемонстрируем результаты работы алгоритмов. Пусть $n = 2, p = (0.1, 0.4), T = 5000$. Прежде чем продемонстрировать результаты, отметим, что для классического варианта тестирования (когда происходит разделение аудитории на две равные группы и каждой группе показывается своя страница) конверсия составит

$(0.1 + 0.4)/2 = 0.25$. Как мы увидим ниже, все алгоритмы дают лучший результат.

В таблице 1 представлены значения конверсий, достигнутых каждым алгоритмом в ходе тестирования. В данной таблице зафиксированы значения для $1000, 3000$ и 5000 показов. Кроме того, на рис. 2 в виде графиков продемонстрированы конверсии алгоритмов в каждый момент времени $t = 1, \ldots, 5000$. Видно, что наименьшую конверсию достигает алгоритм play-the-winner (0.28). Конверсия алгоритма ε-greedy (0.386) немного меньше конверсий, достигаемых другими алгоритмами, впрочем, не исключено, что данный результат можно улучшить путем изменения параметра ε. Результаты остальных алгоритмов примерно одинаковы. Графики конверсий дают также представление о скорости обучения алгоритмов.

В таблице 2 указано количество показов каждой страницы при общем числе показов 5000. На рис. 3 представлены графики функций сожаления алгоритмов. Графики функций сожаления алгоритма play-the-winner и алгоритма ε-greedy (с некоторого момента) имеют линейный рост, в то время как функции сожаления остальных алгоритмов — логарифмический. Логарифмический рост функции сожаления указывает на то, что алгоритм успешно справляется с задачей. Сравнивая графики на рис. 2 и 3, нетрудно заметить, что функция сожаления и величина конверсии связаны обратным соотношением: чем больше конверсия, тем меньше значение функции сожаления.

Рассмотрим теперь случай тестирования 5 страниц. Пусть входные данные имеют следующие значения $n = 5, p = (0.25, 0.3, 0.35, 0.4, 0.45), T = 5000$. Особенностью рассматриваемого случая является то, что конверсии страниц имеют близкие значения.

Результаты тестирования, как и в предыдущем случае, представлены в таблицах 3 и 4 и на рисунках 4 и 5. Здесь наибольшую конверсию имеют ал-

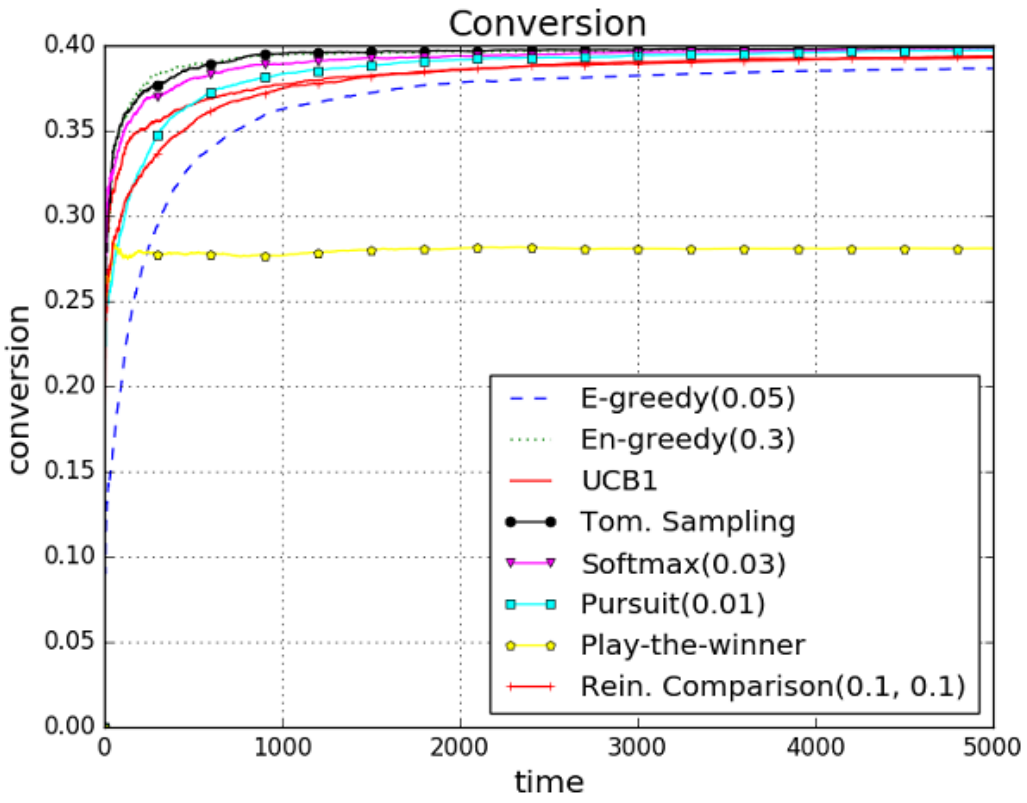

Рис. 2: Конверсии в ходе тестирования ($n = 2$)

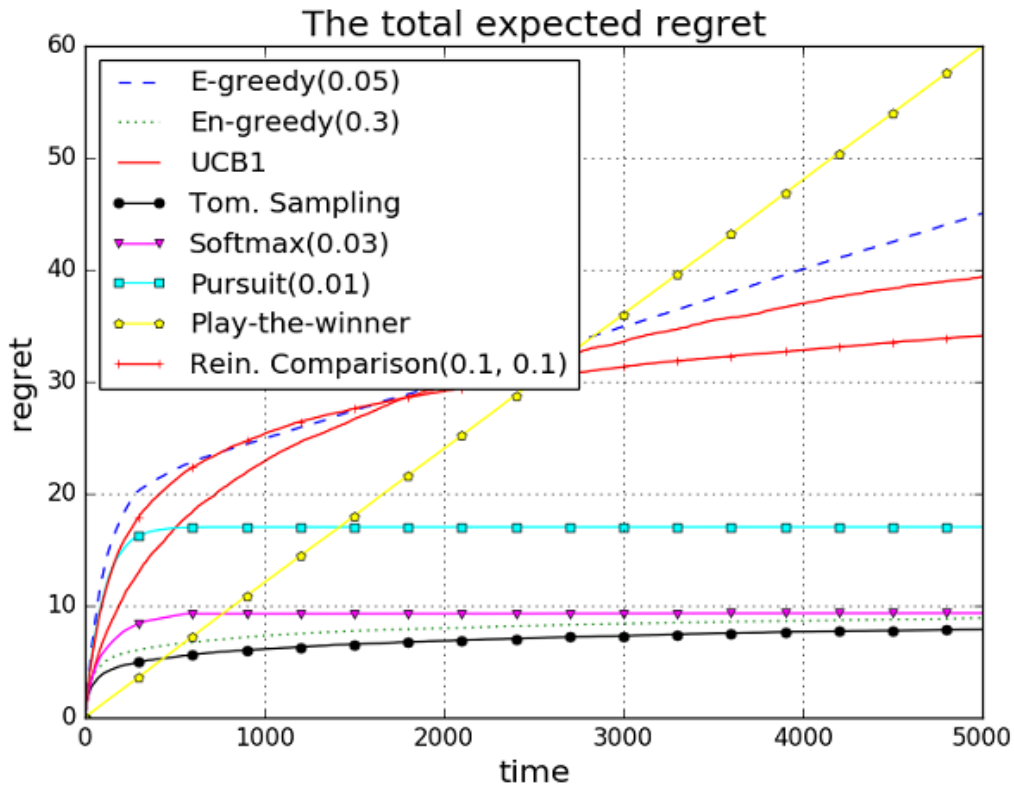

Рис. 3: Функции сожаления ($n = 2$)

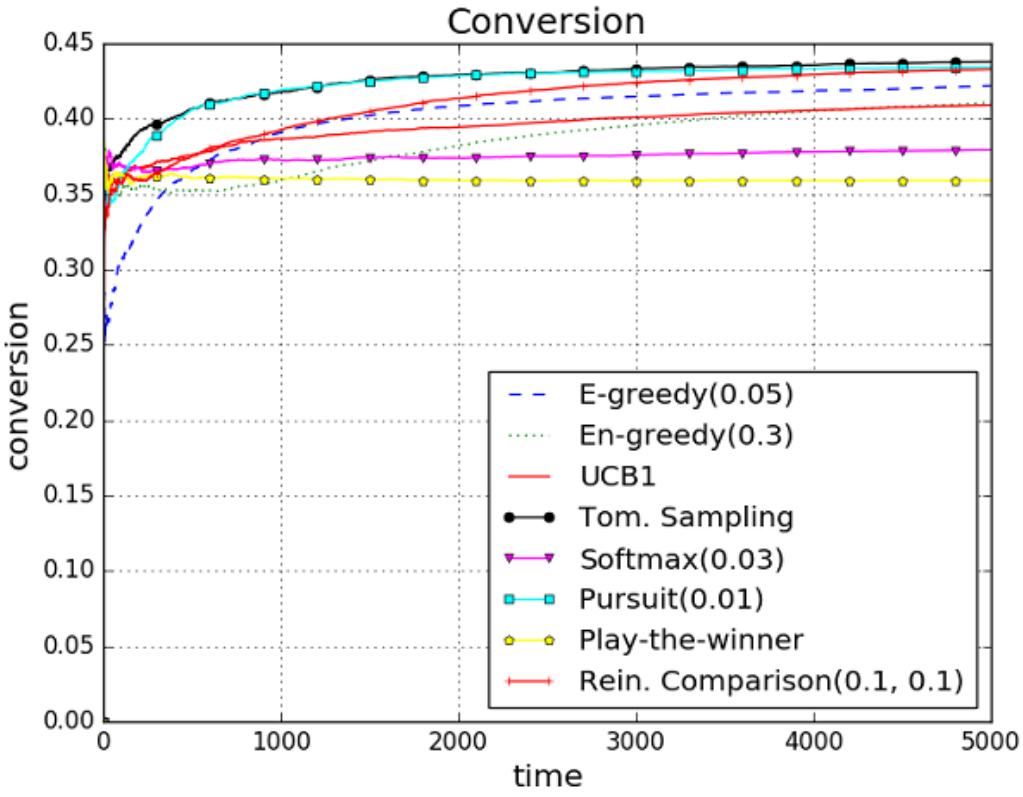

Рис. 4: Конверсии в ходе тестирования ($n = 5$)

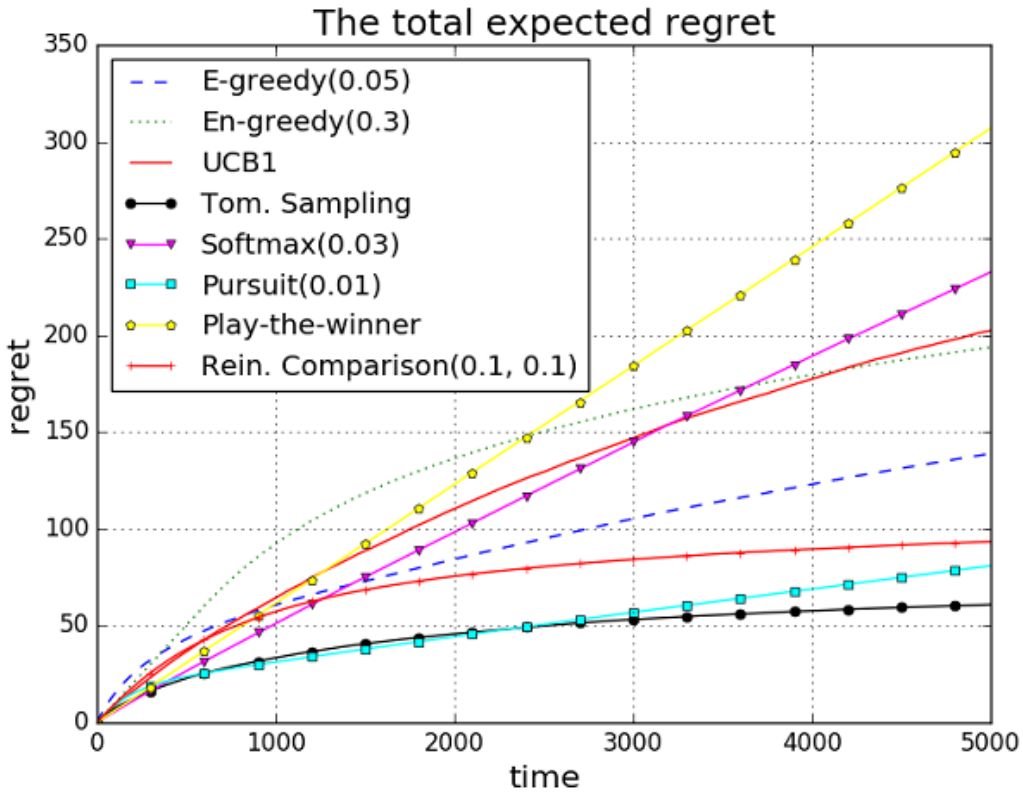

Рис. 5: Функции сожаления ($n = 5$)

Таблица 1: Конверсия в ходе тестирования ($n = 2$)

Алгоритм \ Показы	**1000**	**3000**	**5000**
Play-the-winner	0.28	0.28	0.28
ε-greedy(0.05)	0.361	0.383	0.386
ε_n-greedy(0.3)	0.394	0.398	0.399
Softmax(0.03)	0.392	0.397	0.398
UCB1	0.379	0.389	0.392
Pursuit(0.03)	0.38	0.395	0.397
Reinforcement comparison(0.1, 0.1)	0.376	0.39	0.393
Thompson Sampling	0.395	0.397	0.399

Таблица 2: Количество показов ($n = 2$)

Алгоритм \ Страницы	**1**	**2**
Play-the-winner	2003	2997
ε-greedy(0.05)	226	4774
ε_n-greedy(0.3)	34	4966
Softmax(0.03)	41	4959
UCB1	133	4867
Pursuit(0.03)	57	4943
Reinforcement comparison(0.1, 0.1)	116	4884
Thompson Sampling	25	4975

горитмы Thompson Sampling (0.439), pursuit (0.436) и reinforcement comparison (0.433). Как и ранее, наименьшая конверсия (0.359) соответствует алгоритму play-the-winner. Следует также отметить низкую конверсию (0.383) алгоритма softmax, хотя в предыдущем случае он имел один из лучших результатов.

Таблица 3: Конверсия в ходе тестирования ($n = 5$)

Алгоритм \ Показы	**1000**	**3000**	**5000**
Play-the-winner	0.358	0.359	0.359
ε-greedy(0.05)	0.39	0.412	0.424
ε_n-greedy(0.3)	0.358	0.395	0.412
Softmax(0.1)	0.371	0.383	0.383
UCB1	0.385	0.402	0.41
Pursuit(0.03)	0.415	0.43	0.436
Reinforcement comparison(0.1, 0.1)	0.396	0.422	0.433
Thompson Sampling	0.413	0.433	0.439

Таблица 4: Количество показов ($n = 5$)

Алгоритм \ Страницы	**1**	**2**	**3**	**4**	**5**
Play-the-winner	859	912	997	1065	1167
ε-greedy(0.05)	152	90	343	1083	3332
ε_n-greedy(0.3)	377	378	378	594	3273
Softmax(0.1)	370	661	980	1763	1226
UCB1	223	339	515	1072	2851
Pursuit(0.03)	32	39	109	1084	3736
Reinforcement comparison(0.1, 0.1)	118	149	212	551	3970
Thompson Sampling	50	73	147	430	4300

2.6. Выводы

Исходя из результатов симуляции, можно сделать вывод, что использование алгоритмов для решения задачи о многоруком бандите увеличивает общую конверсию за время тестирования (в сравнении с классическим способом тестирования).

При использовании рассматриваемого подхода к тестированию реальных интернет-страниц возникает задача выбора подходящего алгоритма. Из

результатов симуляции тестирования понятно, что производительность алгоритмов зависит от входных данных, что усложняет выбор алгоритма. Поэтому при выборе стратегии мы рекомендуем в первую очередь отталкиваться от теоретической оценки функции сожаления. Из рассмотренных нами алгоритмов только UCB1, ε_n-greedy и Thompson Sampling имеют такие оценки.

3. Глава 2. Тестирование интернет-страниц на основе задачи о контекстном бандите

В данной главе рассматривается приложение *задачи о контекстном бандите* для тестирования интернет-страниц. В начале главы формулируются задачи. Во втором параграфе дается формальное определение задачи о контекстном бандите. Третий параграф главы содержит описание алгоритма LinUCB, используемого нами в численном эксперименте. В четвертом параграфе предложен подход к тестированию интернет-страниц с помощью алгоритмов для решения задачи о контекстном бандите. Проведен численный эксперимент, состоящий в программной симуляции тестирования. Продемонстрированы результаты работы алгоритма LinUCB. Параграф «выводы» завершает главу.

3.1. Постановка задачи

В реальных ситуациях в условиях задачи о многоруком бандите кроме значений полученных выигрышей от выбора альтернатив часто доступна некоторая дополнительная информация, которую можно использовать для улучшения выбора. Так, в тестировании интернет-страниц в качестве такой информации можно рассматривать источник перехода пользователя. В этом случае тестирование можно свести к задаче о контекстном бандите. Задачи настоящей главы: предложить подход к тестированию, в котором учитывается источник перехода пользователя, разработать программную симуляцию тестирования, провести анализ результатов численного эксперимента.

3.2. Задача о контекстном бандите

Одной из многочисленных модификаций задачи о многоруком бандите является так называемая задача о контекстном бандите (contextual bandit problem) [15].

В задаче о контекстном бандите игрок также многократно выбирает действия, получая за это награду, однако теперь получаемый выигрыш зависит от некоторого «контекста». Таким образом, игрок, делая выбор, должен учитывать поступающую контекстную информацию. Задачей игрока, как и прежде, является получение наибольшего суммарного выигрыша за время T. Перейдем к формальному описанию задачи.

Обозначим через A множество из n выбираемых действий $a_1, \ldots, a_n$. В каждый момент времени $t = 1, \ldots, T$ игрок, как и в задаче о многоруком бандите, выбирает очередное действие. Однако теперь перед каждым выбором он наблюдает n векторов $x_{t,a} \in \mathbb{R}^d, a \in A$. Вектора $x_{t,a}, a \in A$ называются *контекстными*, их содержательный смысл зависит от конкретной задачи. Обозначим множество контекстных векторов в момент времени t через x_t и для краткости будет называть *контекстом*. Как и ранее, игрок, выбирая некоторое действие $a_{j(t)}$, получает выигрыш $r_{a_{j(t)}}$ — реализация случайной величины $\xi_{j(t)}$. В отличии от классической задачи о многоруком бандите здесь распределения случайных величин $\xi_1, \ldots, \xi_n$ зависят от контекстных векторов. Задача игрока — максимизировать суммарный выигрыш

$$E\Big[\sum_{t=1}^{T} r_{a_{j(t)}}\Big],$$

что эквивалентно минимизации *функции сожаления*

$$R(T) = E\Big[\sum_{t=1}^{T} r_{a^*_{j(t)}}\Big] - E\Big[\sum_{t=1}^{T} r_{a_{j(t)}}\Big],$$

где $a^*_{j(t)}$ — действие с максимальным ожидаемым выигрышем в момент времени t.

Алгоритмы, изученные нами в первой главе, не применимы для решения описанной выше задачи, поскольку не учитывают зависимость выигрышей от контекста. Алгоритмы, используемые для решения задачи о контекстном бандите, в каждый момент времени t выбирают действие, основываясь на ранее полученных данных вида

$$(x_{1,a_{j(1)}}, a_{j(1)}, r_{a_{j(1)}}), \ldots, (x_{t-1,a_{j(t-1)}}, a_{j(t-1)}, r_{a_{j(t-1)}})$$

и значениях текущего контекста x_t.

Литература, посвященная задаче о контекстном бандите, изобилует алгоритмами по её решению. Наиболее популярны LinUCB [21], Epoch-greedy [20], LinRel [12], Thompson Sampling [9].

На практике задача о контекстном бандите применяется чаще, чем классическая задача о многоруком бандите, поскольку реальные процессы без контекстной информации встречаются довольно редко. Перечислим некоторые приложения задачи о контекстном бандите: персонализированная рекомендация новостных статей [21], публикация контента в социальных сетях [18], динамическая корректировка цен в интернет-магазинах [25], выбор рекламных баннеров [7] и т. д.

3.3. Алгоритм LinUCB

Алгоритм LinUCB (Linear Upper Confidence Bound), предложенный в работе [21], является расширением алгоритма UCB1 для задачи о контекстном бандите. Пусть в каждый момент времени t действию $a \in A$ соответствует контекстный вектор $x_{t,a} \in \mathbb{R}^d$.

Основное предположение алгоритма заключается в том, что математическое ожидание выигрыша $r_{t,a}$ в результате выбора действия a линейно зависит от его контекстного вектора $x_{t,a}$, т. е:

$$E[r_{t,a}|x_{t,a}] = x_{t,a}^T \theta_a^*,$$

где θ_a^* — неизвестный вектор коэффициентов.

Подчеркнем, что данное предположение не является особенностью рассматриваемого алгоритма. Задачи, в которых принимается такое допущение, называются *задачами о контекстном бандите с линейными функциями выигрыша* [15]. Данный подход впервые предложен в работе [7].

Пусть некоторое действие a было выбрано m раз. Обозначим через D_a матрицу размерности $m \times d$, состоящую из m предыдущих контекстных векторов $x_{t,a}$ действия a. Через $b_a \in \mathbb{R}^m$ обозначим вектор выигрышей, в результате предыдущих выборов действия a. Используя гребневую регрессию (ridge регрессию) для данных (D_a, b_a), построим оценку вектора θ_a^*:

$$\widehat{\theta}_a = (D_a^T D_a + I_a)^{-1} D_a^T b_a,$$

где I_d — квадратная единичная матрица размерности d.

В работе [31] доказывается, что для выигрышей $r_{t,a}$, являющихся реализациями независимых случайных величин с математическим ожиданием $E[r_{t,a}|x_{t,a}] = x_{t,a}^T \theta_a^*$ с вероятностью $1 - \delta$ выполняется неравенство:

$$\left| x_{t,a}^T \widehat{\theta}_a - E[r_{t,a}|x_{t,a}] \right| \leq \alpha \sqrt{x_{t,a}^T (D_a^T D_a + I_a)^{-1} x_{t,a}},$$

где $\alpha = \sqrt{\frac{1}{2} \ln \frac{2}{\delta}}$.

Обозначим $A_a = D_a^T D_a + I_d$. Алгоритм LinUCB (как и все алгоритмы семейства UCB) выбирает действие с наибольшим значением правой границы доверительного интервала, т. е.

$$a_t = \arg\max_{a \in A} \left(x_{t,a}^T \widehat{\theta}_a + \alpha \sqrt{x_{t,a}^T A_a^{-1} x_{t,a}} \right).$$

Согласно анализу, проведенному в работе [15], для функции сожаления $R(T)$ алгоритма LinUCB с вероятностью $1 - \delta$ справедлива оценка:

$$R(T) = O\Big(\sqrt{Td \ln^3(nT \ln(T)/\delta)} \Big).$$

Там же найдена нижняя оценка функции сожаления:

$$R(T) = \Omega(\sqrt{Td}).$$

3.4. Численный эксперимент

Вновь обратимся к тестированию интернет-страниц. Пусть в тестировании участвуют n страниц. Предположим, что основным каналом, по которому посетители переходят на сайт, является контекстная реклама [4]. Рекламная кампания состоит из рекламных объявлений, кликая по которым пользователи попадают на сайт. Допустим, что рекламная кампания содержит m объявлений $ad_1, \ldots, ad_m$. Будем считать, что каждый посетитель тестируемой страницы попал на сайт, перейдя по одному из m рекламных объявлений.

Вероятна такая ситуация, в которой величина конверсии страницы зависит от рекламного объявления. То есть одна и та же версия тестируемой страницы может иметь разную конверсию для каждого из m рассматриваемых рекламных объявлений. Тогда при тестировании с использованием задачи о многоруком бандите для достижения наибольшей суммарной конверсии следует учитывать номер рекламного объявления. Таким образом, тестирование веб-страниц можно представить как задачу о контекстном бандите, в которой в качестве контекста выступают рекламные объявления.

Как и в главе 1 вместо тестирования реальных страниц продемонстрируем результаты работы алгоритма LinUCB на сгенерированных данных. Для этой цели была разработана программная реализация симуляции тестирования страниц на языке Python (см. приложение 2). Одно из отличий данной реализации, от программы, представленной в первой главе, является генерация контекста. Перед тем как алгоритм LinUCB выберет очередную страницу, программа генерирует контекстный вектор вида $x(t) = (0, \ldots, 1, \ldots, 0) \in \mathbb{R}^m$, где единица на j-том месте означает, что очередной пользователь «пришел» с j-го рекламного объявления. Заметим, что в нашем случае контекстный вектор не зависит от страницы. Напомним, что в предложенной симуляции тестирования при выборе страницы с номером i программа генерирует выигрыш как реализацию случайной величины, имеющей распределение Бернулли с математическим ожиданием p_i, которое соответствует конверсии страницы.

На вход программе подаются число страниц n, число рекламных объявлений m, m векторов вида $p^j = (p_1^j, \ldots, p_n^j), j = 1, \ldots, m$, соответствующих конверсиям страниц для каждого рекламного объявления, вектор вероятностей для генерации контекста $\widehat{p} = (\widehat{p}_1, \ldots, \widehat{p}_m), \sum_{k=1}^{m} \widehat{p}_k = 1$ и число показов T.

Генерация контекста (выбор рекламного объявления) происходит согласно вероятностям вектора $\widehat{p}$. Компонента $\widehat{p}_i$ является вероятностью события «пользователь перешел на сайт с объявления ad_i». Сумма компонент равна единице, поскольку каждый пользователь переходит на сайт с одного из m рекламных объявлений.

В качестве результатов программа выводит конверсию за время тестирования, конверсии страниц для каждого рекламного объявления, количество показов страниц для каждого объявления, число переходов с рекламных объявлений, номер страницы с наибольшей конверсией, а также строит график функции сожаления алгоритма LinUCB и график достигнутой конверсии в хо-

де тестирования.

Рассмотрим сначала случай тестирования трех страниц с двумя рекламными объявлениями. Пусть $n = 3, m = 2, T = 5000$ и

$$p^1 = (0.6, 0.1, 0.5), \quad p^2 = (0.2, 0.7, 0.4), \quad \widehat{p} = (0.5, 0.5).$$

Приведем результаты работы программы. Итак, за время $T = 5000$ с первого рекламного объявления перешел 2501 пользователь, со второго — 2499. Данный результат справедлив, поскольку вероятности переходов с каждого из двух объявлений равны 0.5. Конверсия за время тестирования составила 0.635. В таблице 5 приведены значения числа показов страниц для каждого объявления. В таблице 6 — конверсии страниц, которые, как и полагается, близки значениям векторов p^1 и p^2. На рис. 6 приведен график функции сожаления алгоритма LinUCB, а на рис. 7 — достигаемая конверсия в ходе тестирования. Видно, что функция сожаления имеет логарифмический рост.

Таблица 5: Число показов ($n = 3, m = 2$)

Объявления \ Страницы	**1**	**2**	**3**
ad_1	2133	30	338
ad_2	29	2408	62

Таблица 6: Конверсия страниц ($n = 3, m = 2$)

Объявления \ Страницы	**1**	**2**	**3**
ad_1	0.6	0.09	0.49
ad_2	0.17	0.7	0.36

Как же определить страницу с наибольшей конверсией? Для этого матрицу полученных конверсий страниц необходимо перемножить на вектор со

значениями переходов с рекламных объявлений $b = (2501, 2499)$:

$$(2501, 2499) \cdot \begin{pmatrix} 0.6 & 0.09 & 0.49 \\ 0.17 & 0.7 & 0.36 \end{pmatrix} = (1925.43, 1974.39, 2125.13).$$

Компонента с номером i полученного вектора интерпретируется как количество достигнутых целей при условии показа только i-ой страницы. Поэтому, разделив каждую компоненту данного вектора на общее число переходов 5000, получим вектор со значениями конверсий каждой страницы $(0.385, 0.395, 0.425)$. Таким образом, наибольшую конверсию имеет страница с номером 3.

Если, например, $\widehat{p} = (0.2, 0.8)$, тогда в результате работы программы получим $b = (997, 4003)$. В этом случае уже вторая страница будет иметь наибольшую конверсию.

Рассмотрим еще один пример входных данных. Пусть $n = 3, m = 5$,

$$p^1 = (0.3, 0.4, 0.5), \quad p^2 = (0.4, 0.1, 0.3), \quad p^3 = (0.6, 0.7, 0.5),$$

$$p^4 = (0.15, 0.2, 0.25), \quad p^5 = (0.3, 0.4, 0.5), \quad \widehat{p} = (0.2, 0.2, 0.2, 0.2, 0.2).$$

В результате работы программы конверсия за время тестирования составила 0.44. Наибольшую конверсию имеет страница номер 3. Пользователи равномерно распределяются по рекламным объявлениям. В таблице 7 приведено число показов каждой страницы с учетом рекламных объявлений. Очевидно, что страницы с высокой конверсией показываются большее число раз, чем страницы с низкой конверсией. На рис. 8 продемонстрирован график функции сожаления алгоритма. Сравнивая его с графиком 6, можно видеть более быстрый рост функции сожаления. Это связано в первую очередь с тем, что конверсии страниц имеют близкие значения. На рис. 9 продемонстрирован график достигаемой конверсии в ходе тестирования.

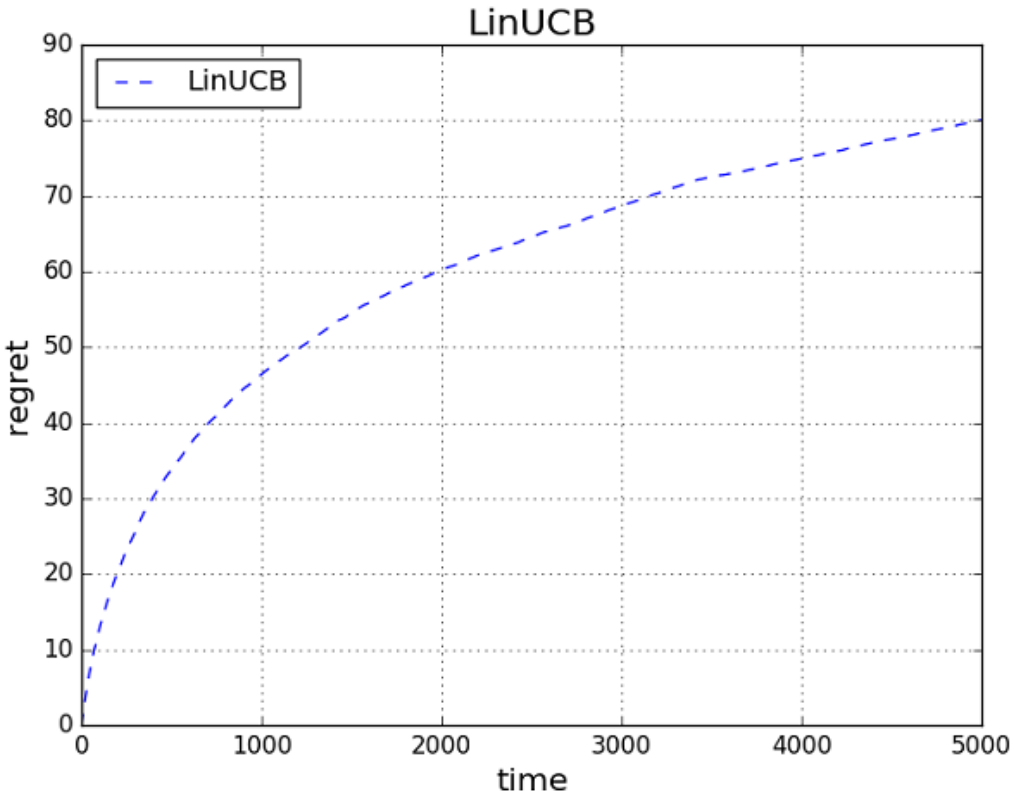

Рис. 6: Функция сожаления ($n = 3, m = 2$)

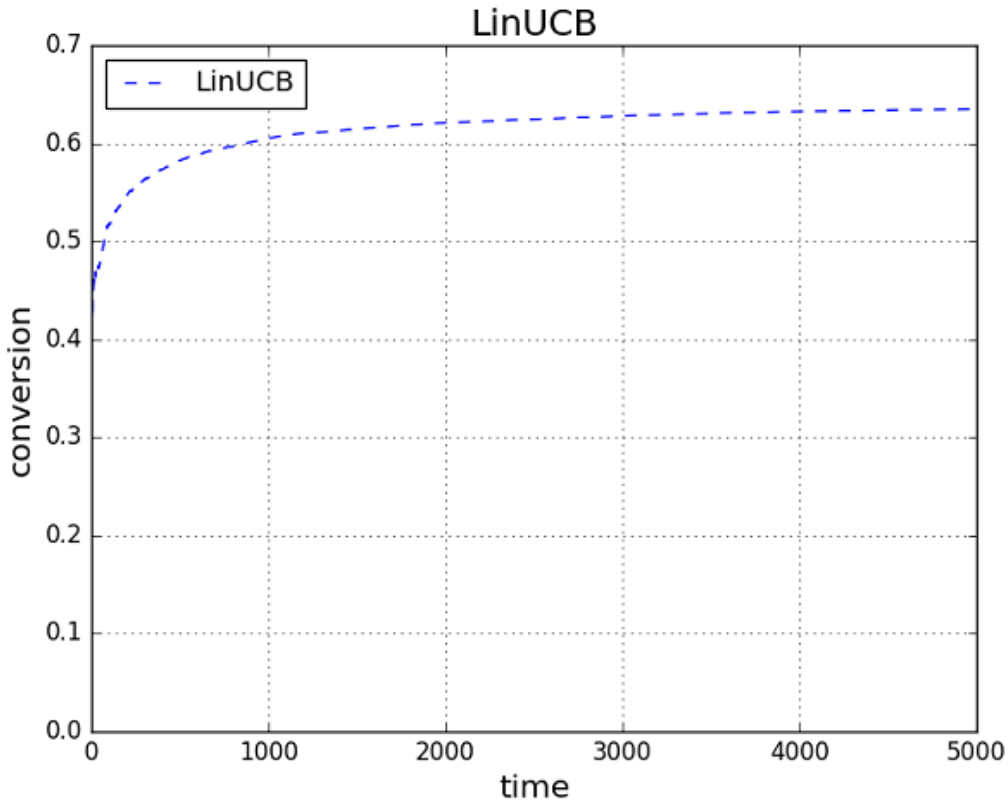

Рис. 7: Конверсия в ходе тестирования ($n = 3, m = 2$)

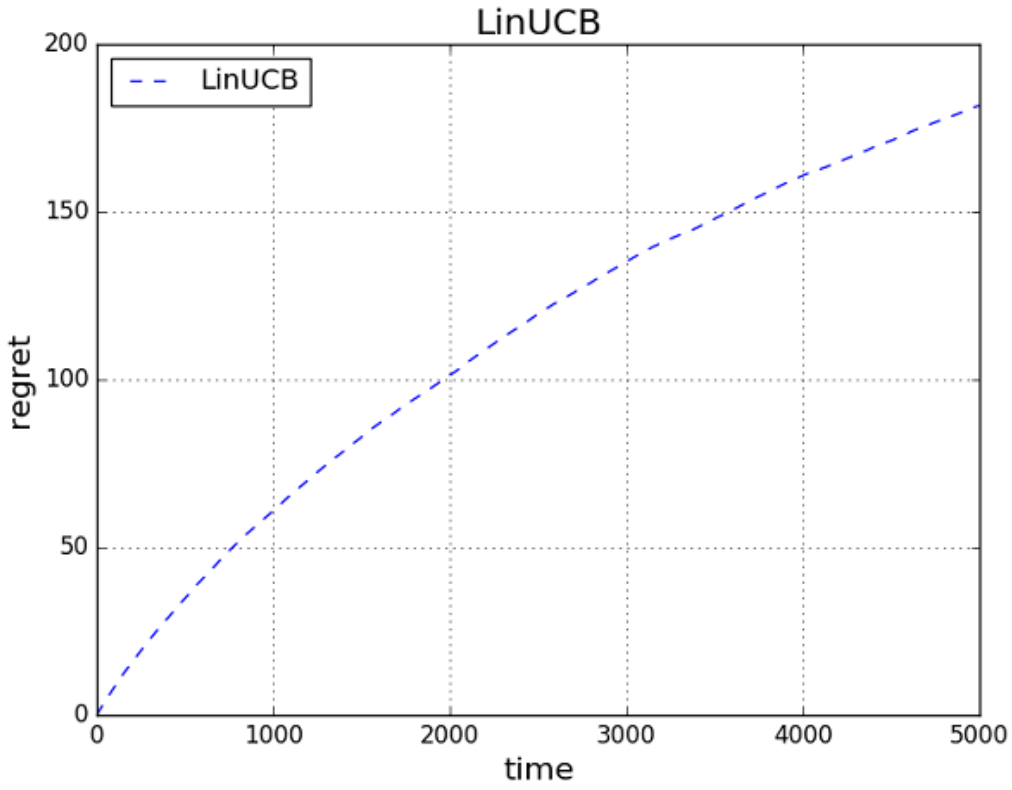

Рис. 8: Функция сожаления ($n = 3, m = 5$)

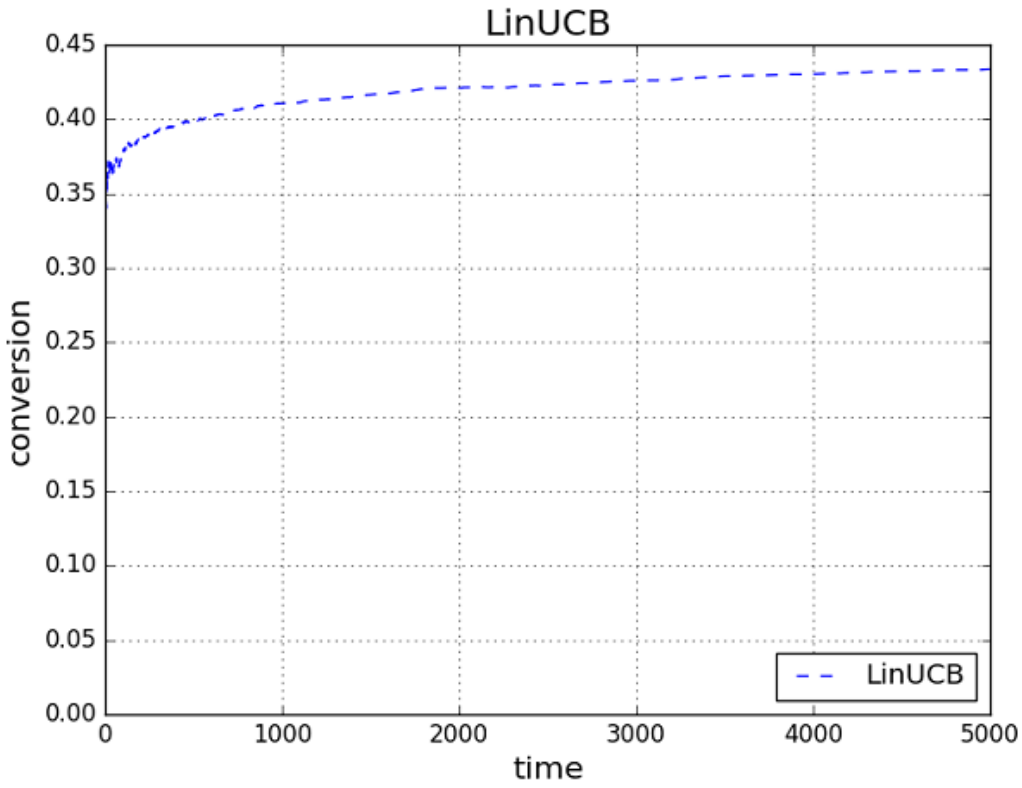

Рис. 9: Конверсия в ходе тестирования ($n = 3, m = 5$)

Таблица 7: Число показов ($n = 3, m = 5$)

Объявления \ Страницы	**1**	**2**	**3**
ad_1	92	211	693
ad_2	743	53	202
ad_3	201	708	92
ad_4	177	286	541
ad_5	85	195	721

3.5. Выводы

В качестве вывода, на простом примере продемонстрируем преимущество данного подхода в сравнении с подходом, предложенным в главе 1. Пусть в тестировании участвуют три страницы с двумя рекламными объявлениями, причем одна половина пользователей переходит с одного объявления, а другая половина — со второго. В наших обозначениях $n = 3, m = 2, \widehat{p} = (0.5, 0.5)$. И пусть конверсии страниц имеют следующие значения:

$$p^1 = (0.1, 0.2, 0.3), \quad p^2 = (0.5, 0.4, 0.2).$$

Алгоритм LinUCB в этом случае достигает конверсию за время тестирования ($T = 5000$), равную 0.378.

Поскольку вектор $\widehat{p} = (0.5, 0.5)$, то в качестве вектора конверсии для алгоритмов, рассматриваемых в главе 1, следует использовать средний вектор, т. е.

$$p = \left(\tfrac{0.1+0.5}{2}, \tfrac{0.2+0.4}{2}, \tfrac{0.3+0.2}{2}\right) = (0.3, 0.3, 0.25).$$

Очевидно, что с таким вектором p ни один алгоритм не будет достигать конверсию выше 0.3. Таким образом, применение подхода к тестированию, основанного на задаче о контекстном бандите, позволяет увеличить конверсию за время тестирования.

4. Глава 3. Задача о многоруком бандите с экспертами

Глава посвящена одной модификации задачи о многоруком бандите. Начало главы содержит постановку задачи. Во втором параграфе формулируется задача о многоруком бандите при наличии эксперта. Третий параграф главы посвящен модификации алгоритма UCB1. В четвертом параграфе проводится численный эксперимент, в котором демонстрируются результаты применения модифицированного алгоритма UCB1 для задачи о многоруком бандите с одним экспертом. В пятом параграфе рассматривается задача о многоруком бандите с m экспертами. Формулируются три метода формирования единой экспертной подсказки. И, наконец, в шестом параграфе приведены результаты численного эксперимента для задачи о многоруком бандите с m экспертами. В конце главы сформулированы выводы.

4.1. Постановка задачи

Среди контекстной информации можно выделить ту, которая содержит прямые рекомендации по выбору альтернатив. Такие рекомендации будем называть *подсказками эксперта*. Задачи данной главы: формализовать задачу о многоруком бандите с экспертами, модифицировать известный алгоритм для решения задачи о многоруком бандите, провести численный эксперимент.

4.2. Задача о многоруком бандите при наличии эксперта

Вновь вернемся к классической задаче о многоруком бандите (см. п. 1 главы 1) и рассмотрим одну её модификацию. Пусть в дополнении к задаче о многоруком бандите имеется эксперт, который в каждый момент времени t делает предположение о значениях выигрышей действий $a_1, \ldots, a_n$. Каждый раз,

когда игрок выбирает действие, эксперт предлагает вектор $(b_1(t), \ldots, b_n(t))$, в котором компонента $b_i(t)$ есть предположение о значении выигрыша в результате выбора действия a_i в момент времени t. Допустим, что эксперт владеет некоторой информацией о значениях выигрышей выбираемых действий, что позволяет ему с некоторой степенью точности «предсказывать» выигрыши в каждый момент времени.

Формализуем такие подсказки эксперта следующим образом. Пусть компоненты вектора $(b_1(t), \ldots, b_n(t))$ являются реализациями случайных величин $\widehat{\xi}_1, \ldots, \widehat{\xi}_n$, имеющих распределение Бернулли с неизвестными параметрами $\widehat{p}_1, \ldots, \widehat{p}_n$. Здесь рассматривается распределение Бернулли, поскольку случайные величины, соответствующие действиям, имеют распределение Бернулли. Задача игрока состоит в том же, однако теперь он, помимо данных о значениях полученных выигрышей, может воспользоваться подсказками эксперта.

4.3. Модификация алгоритма UCB1

Поставим задачу модифицировать алгоритм UCB1 таким образом, чтобы он учитывал информацию, полученную от эксперта. Изменим алгоритм так, чтобы выбор игрока зависел от значения подсказки эксперта и от её точности.

Для этого добавим в формулу (1) слагаемое

$$\overline{b}_i(t)k_i(t),$$

где $\overline{b}_i(t)$ — среднее значение b_i за время t, т. е. средняя подсказка для действия a_i, а

$$k_i(t) = e^{-|\overline{b}_i(t) - \overline{p}_i(t)|},$$

— величина, характеризующая точность подсказки для действия a_i. Данный коэффициент позволит учитывать неточные подсказки с меньшим весом. Та-

ким образом, модифицированный алгоритм UCB1 на каждом шаге выбирает действие с индексом:

$$j(t) = \arg\max_{i=1,\dots,n} \left(\overline{p}_i + \sqrt{\frac{2\ln t}{n_i}} + \overline{b}_i(t) k_i(t) \right). \quad (2)$$

Проведем краткий анализ добавленного слагаемого $s(t) = \overline{b}(t)k(t)$. На рис. 10 показаны графики $s(t)$ при различных фиксированных значениях $\overline{b}(t)$. Нетрудно заметить, что максимум величины $s(t)$ при фиксированном значении средней подсказки $\overline{b}^*(t)$ достигается при $\overline{p}(t) = \overline{b}^*(t)$ и равен значению средней величины подсказки. Также видно, что $s(t)$ симметрично убывает относительно вершины $\overline{b}^*(t)$. Отсюда следует, что, например, при $\overline{b}(t) = 0.5$ значение $s(t)$ будет одинаковым для $\overline{p}(t) = 0.8$ и $\overline{p}(t) = 0.3$. Такой, казалось бы, противоречивый результат, не должен настораживать читателя, поскольку значение величины среднего выигрыша является первым слагаемым в формуле (2), и, таким образом, при прочих равных условиях, будет выбрано действие с $\overline{p}(t) = 0.8$.

На рис. 11 продемонстрированы графики функции $s(t)$ при различных фиксированных значениях $\overline{p}(t)$. Видно, каким образом возрастает $s(t)$ при фиксированном $\overline{p}(t)$: быстрый рост до значения $\overline{b}(t) = \overline{p}(t)$, затем медленное возрастание до $\overline{b}(t) = 1$.

4.4. Численный эксперимент (один эксперт)

Для анализа результатов работы модифицированного алгоритма UCB1 была выполнена его программная реализация на языке Python (см. приложение 3). На вход программе подается число действий n, вектор математических ожиданий случайных величин, соответствующих выбираемым действиям $(p_1, \dots, p_n)$, вектор математических ожиданий случайных величин, соответствующих подсказкам эксперта $(\widehat{p}_1, \dots, \widehat{p}_n)$ и число итераций T. Програм-

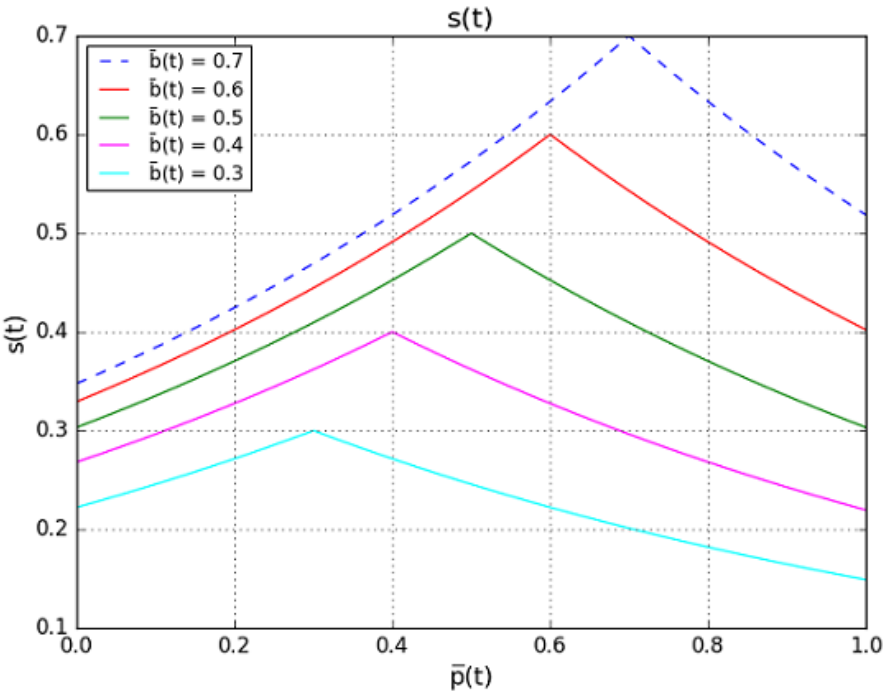

Рис. 10: $s(t)$ при фиксированных $\overline{b}(t)$

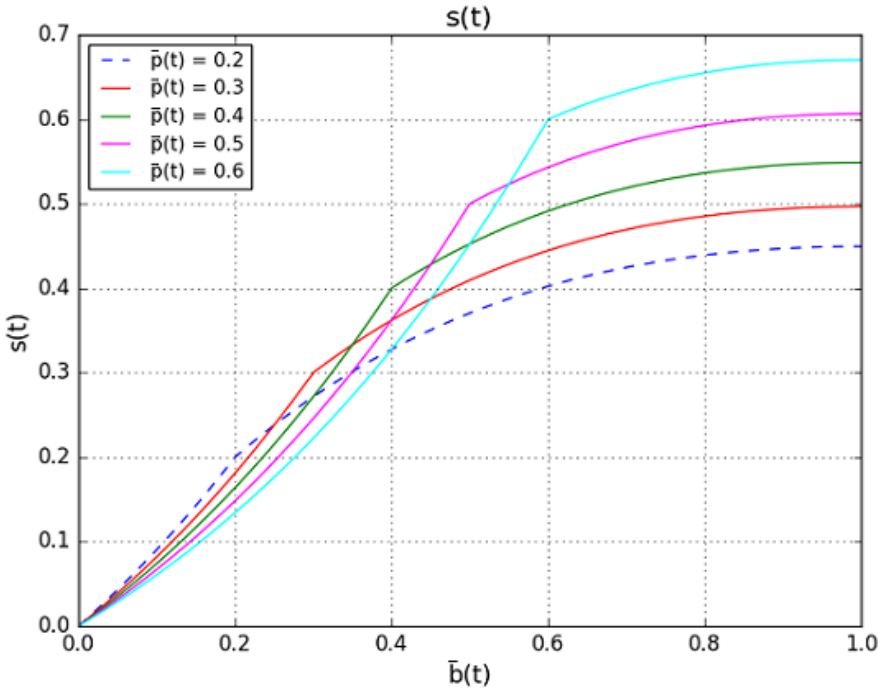

Рис. 11: $s(t)$ при фиксированных $\overline{p}(t)$

ма строит график функции сожаления. Напомним, что функция сожаления и величина выигрыша связаны обратным соотношением: чем меньше значение функции сожаления, тем больше значение выигрыша. Поэтому для сравнения полученных выигрышей достаточно привести графики функций сожаления.

Рассмотрим пример входных данных. Пусть $n = 5, T = 5000$ и

$$(p_1, p_2, p_3, p_4, p_5) = (0.3, 0.45, 0.5, 0.47, 0.1).$$

Для сравнения приведем графики функции сожаления алгоритма UCB1 без учета эксперта и функции сожаления модифицированного алгоритма со следующими значениями математических ожиданий подсказок эксперта $(\widehat{p}_1, \ldots, \widehat{p}_n)$:

$$e_1 = (0.5, 0.45, 0.1, 0.5, 0.7), \quad e_2 = (0.1, 0.1, 0.6, 0.1, 0.1),$$

$$e_3 = (0.3, 0.45, 0.5, 0.47, 0.1), \quad e_4 = (0.1, 0.1, 0.1, 0.1, 0.1),$$

$$e_5 = (0.2, 0.3, 0.45, 0.32, 0.05).$$

Результаты работы программы в виде графиков функций сожаления представлены на рисунке 12. В подписи к графикам указаны вектора математических ожиданий подсказок экспертов. Подпись «UCB1» соответствует функции сожаления алгоритма UCB1 без эксперта. Как видно из рисунка, медленнее всех возрастает функция сожаления, соответствующая вектору математических ожиданий подсказок эксперта e_2. Данное поведение алгоритма объясняется тем, что третья компонента, которая и соответствует действию с наибольшим средним выигрышем, намного больше других. Таким образом эксперт уверен, что третье действие приносит наибольший выигрыш. Максимум функции сожаления достигается для вектора e_1, в котором, наоборот, компоненты, соответствующие неоптимальным действиям, завышены. Второй

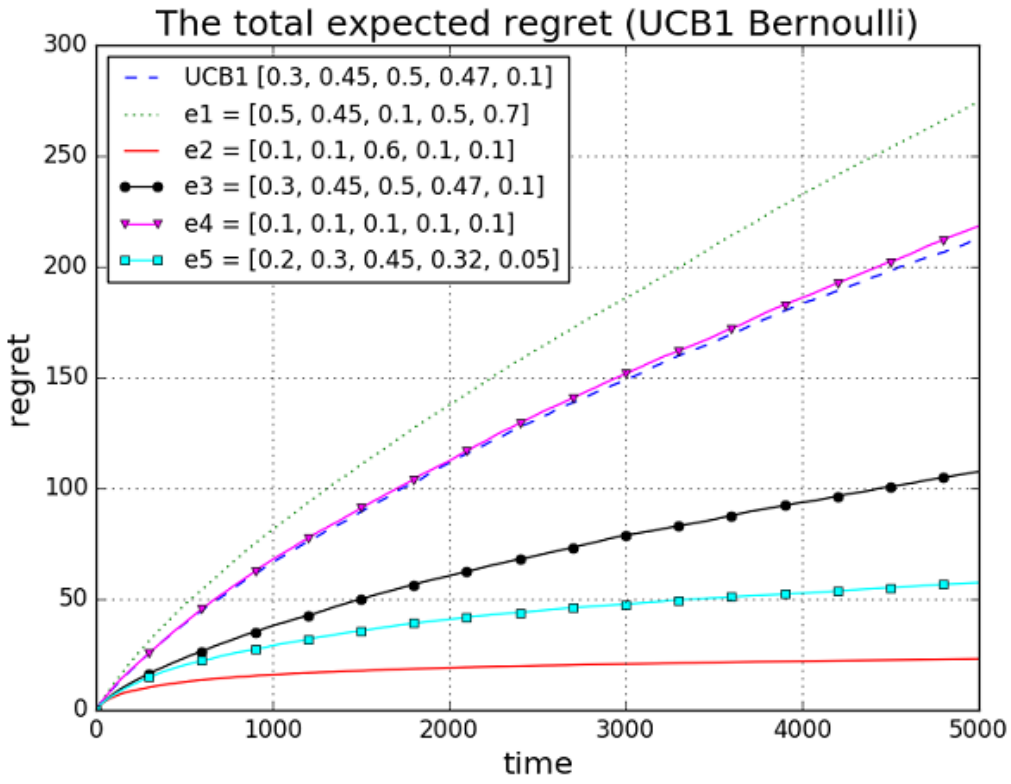

Рис. 12: Функции сожаления ($n = 5, m = 1$)

по величине является функция сожаления, соответствующая вектору e_4, в котором все компоненты равны 0.1. Важно отметить, что подсказки эксперта с высокой степенью точности (вектор e_3) способствуют увеличению выигрыша.

4.5. Задача о многоруком бандите с m экспертами

Рассмотрим задачу о многоруком бандите с m экспертами. Подобно вышеописанному каждый эксперт в момент времени t предлагает вектор подсказок $(b_1(t), \ldots, b_n(t))$. Таким образом, можно сформировать матрицу подсказок в момент времени t

$$B(t) = \begin{pmatrix} b_{11}(t) & b_{12}(t) & \dots & b_{1n}(t) \\ b_{21}(t) & b_{22}(t) & \dots & b_{2n}(t) \\ \dots & \dots & \dots & \dots \\ b_{m1}(t) & b_{m2}(t) & \dots & b_{mn}(t) \end{pmatrix},$$

где строка $b_i(t) = (b_{i1(t)}, \dots, b_{in}(t))$ есть вектор подсказок i-го эксперта в момент времени t.

Как и ранее, подсказка $b_{ij}(t)$ есть реализация распределенной по Бернулли случайной величины $\widehat{\xi}_{ij}$. Задача для игрока остается прежней, однако теперь он может принимать решения, основываясь на подсказках каждого из m экспертов. Матрицу средних подсказок в момент времени t будем обозначать через $\overline{B}(t)$.

К решению задачи о многоруком бандите с m экспертами можно подойти двумя путями. Первый заключается в модификации алгоритма (например, UCB1) для случая с m экспертами. Другой способ состоит в том, чтобы каким-либо образом извлекать из матрицы подсказок вектор и применять уже модифицированный алгоритм UCB1. Второй способ нам кажется интересным, поэтому им и воспользуемся. Рассмотрим различные способы формирования вектора из матрицы подсказок экспертов. Вектор, рассматриваемый в качестве подсказок в момент времени t, будем обозначать $(\widetilde{b}_1(t), \dots, \widetilde{b}_n(t))$.

Как уже было замечено, точные подсказки эксперта увеличивают выигрыш. На этом замечании построены первые два метода формирования вектора подсказок.

Метод 1. Каждый раз будем выбирать вектор подсказок эксперта, для которого достигается минимум евклидовой метрики:

$$\rho(\overline{b}_i(t), \overline{p}(t)) = \sqrt{\sum_{j=1}^{n} \left(\overline{b}_{ij}(t) - \overline{p}_j(t)\right)^2}.$$

Здесь $\overline{b}_i(t) = (\overline{b}_{i1}(t), \ldots, \overline{b}_{in}(t))$ есть средний вектор подсказок i-го эксперта за время t, а $\overline{p}(t) = (\overline{p}_1(t), \ldots, \overline{p}_n(t))$ — вектор средних выигрышей за время t.

Метод 2. В каждом столбце матрицы средних подсказок экспертов $\overline{B}(t)$ будем выбирать компоненту, для которой достигается минимум:

$$\widetilde{b}_j(t) = \min_{i=1,\ldots,m} \left|\overline{b}_{ij}(t) - \overline{p}_j(t)\right|, j = 1, \ldots, n.$$

Метод 3. Здесь в каждый момент времени t будем определять лучшего эксперта и рассматривать его вектор подсказок. Под лучшим экспертом будем понимать эксперта, чьи подсказки приносят наибольший выигрыш. Поскольку выбор эксперта влияет на выигрыш игрока, а его подсказки являются случайными величинами, то поиск наилучшего эксперта можно рассмотреть как еще одну задачу о многоруком бандите. Для решения этой задачи можно воспользоваться любым алгоритмом независимо от того, какой алгоритм используется для выбора действия. Пусть, например, для выбора эксперта используется все тот же алгоритм UCB1. Тогда игрок в каждый момент времени t выбирает эксперта с номером $j(t)$, где

$$j(t) = \arg\max_{i=1,\ldots,m} \left(\overline{r}_i + \sqrt{\tfrac{2\ln t}{m_i}}\right),$$

где $\overline{r}_i$ — средний выигрыш в результате выбора i-го эксперта, а m_i — число раз, когда выбирался эксперт с номером i.

4.6. Численный эксперимент (m экспертов)

Для проведения эксперимента была модифицирована программа из п. 4.3. На вход программе подается число действий n, вектор математиче-

ских ожиданий случайных величин, соответствующих выбираемым действиям $(p_1, \ldots, p_n)$, матрица $\widehat{P}$ размерности $m \times n$ математических ожиданий случайных величин, соответствующих подсказкам эксперта (i-ая строка — подсказки i-го эксперта) и число игр T. Программа строит график функции сожаления. Программный код методов приведен в приложении 4. Полностью код программы доступен в [3].

Пусть $n = 5, p = (0.3, 0.45, 0.6, 0.4, 0.1), T = 1000$ и матрица математических ожиданий подсказок экспертов имеет вид

$$\widehat{P} = \begin{pmatrix} \mathbf{0.3} & 0.1 & 0.1 & 0.1 & 0.1 \\ 0.1 & \mathbf{0.45} & 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & \mathbf{0.6} & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 & \mathbf{0.4} & 0.1 \\ 0.1 & 0.1 & 0.1 & 0.1 & \mathbf{0.1} \end{pmatrix}$$

На рис. 13 продемонстрированы графики функции сожаления каждого из трех методов формирования вектора подсказок. Таким образом, в данном случае наименьшее значение функции сожаления соответствует первому методу, а наибольшее — третьему. В таблице 8 показано, сколько раз выбирался каждый эксперт с помощью первого и третьего метода (второй метод эксперта не выбирает). В первом методе подавляющее число раз выбирался третий эксперт, хотя на первый взгляд кажется, что каждый эксперт имеет одинаковую вероятность быть выбранным. Такое поведение алгоритма объясняется тем, что третье действие (с математическим ожиданием 0.6) выбирается чаще всего, поэтому и средний выигрыш его близок к значению 0.6 (закон больших чисел [1]). Средние подсказки экспертов обновляются каждый раз, поэтому их значения одинаково сходятся к значениям математических ожиданий. А поскольку математические ожидания на диагонали матрицы $\widehat{P}$ совпадают с эле-

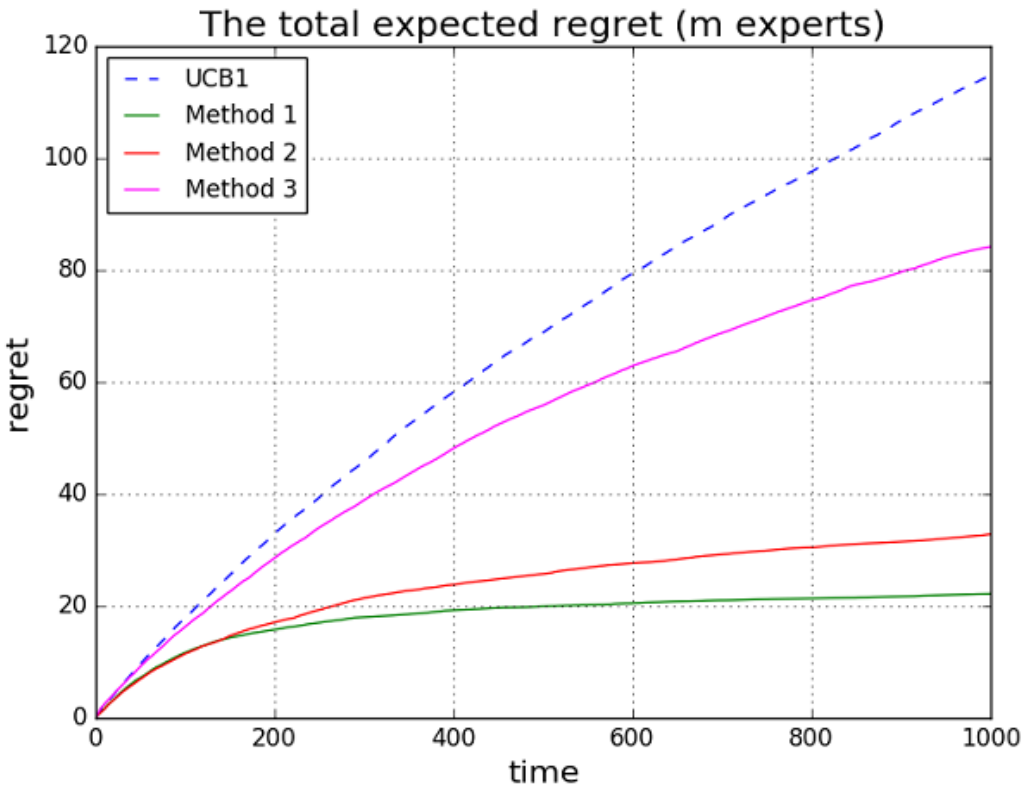

Рис. 13: Функции сожаления ($n = 5, m = 5$)

ментами вектора p и все остальные значения матрицы равны 0.1, то, принимая во внимание вышесказанные рассуждения, можно заключить, что евклидова метрика с наибольшей вероятностью будет минимальна для третьего эксперта.

Каждый элемент матрицы N равен числу выборов подсказки во втором методе. Таким образом, как это и следует из вида входных данных, подсказки, стоящие на диагонали, выбирались чаще всего (за исключением последнего столбца).

$$N = \begin{pmatrix} \mathbf{652} & 48 & 17 & 52 & 192 \\ 89 & \mathbf{816} & 13 & 49 & 179 \\ 91 & 49 & \mathbf{964} & 60 & 194 \\ 118 & 36 & 4 & \mathbf{806} & 224 \\ 50 & 51 & 2 & 33 & 211 \end{pmatrix}$$

Таблица 8: Число выборов экспертов ($n = 5, m = 5$)

Метод \ Эксперт	**1**	**2**	**3**	**4**	**5**
Метод 1	7	30	934	27	2
Метод 3	68	152	619	111	50

4.7. Выводы

Результаты численных экспериментов соответствуют интуитивным представлениям о влиянии подсказок эксперта. Точные подсказки способствуют увеличению выигрыша. В большинстве запусков методы 1 и 2 давали результат лучше третьего. В дальнейшем, конечно, необходимо провести анализ функции сожаления модифицированного алгоритма.

Задача о многоруком бандите с экспертами может быть применена для тестирования интернет-страниц. Экспертную подсказку, например, можно построить на основе средней продолжительности визита пользователя. Как правило, при прочих равных условиях, высокое значение средней продолжительности визита свидетельствует о высокой конверсии страницы. Следовательно, подсказки можно генерировать на основе средней продолжительности визита: чем выше значение, тем с большей вероятностью прогнозировать достижение конверсионной цели.

5. Заключение

В заключении сформулируем основные результаты.

Проведен обзор алгоритмов для решения стохастической задачи о многоруком бандите. Тестирование интернет-страниц свелось к задаче о многоруком бандите. Разработана программная реализация симуляции тестирования. Приведены результаты тестирования страниц с помощью 8-ми алгоритмов. На основании результатов можно утверждать, что использование алгоритмов для решения задачи о многоруком бандите в тестировании страниц увеличивает конверсию за время тестирования.

Предложен новый подход к тестированию интернет-страниц, основанный на задаче о контекстном бандите. Данный способ тестирования позволяет учитывать источник перехода пользователя. А именно в настоящей работе рассматриваются переходы с рекламных объявлений. С учетом нового подхода разработана программная реализация симуляции тестирования, в основе которой лежит алгоритм LinUCB. Конверсия, получаемая в результате применения данного подхода, превосходит результат тестирования на основе задачи о многоруком бандите.

Предложена модификация задачи о многоруком бандите, которая заключается в добавлении экспертных подсказок. Сначала формулируется задача о многоруком бандите с одним экспертом. Для её решения произведена модификация алгоритма UCB1. Затем задача обобщается на случай m экспертов. Предложены три метода формирования единой экспертной подсказки. Разработана программа, с помощью которой произведен численный эксперимент. На конкретных входных данных показано влияние экспертных подсказок на значение выигрыша.

Список литературы

[1] Буре В. М., Парилина Е. М. Теория вероятностей и математическая статистика. М.: Лань, 2013. 416 с.

[2] Лазутченко А. Н. О робастном управлении в случайной среде, характеризуемой нормальным распределением доходов с различными дисперсиями // Труды Карельского научного центра Российской академии наук. 2015. №. 10.

[3] Программный код для ВКР. URL: `https://github.com/patrickjsmirnov/vkr_master` (дата обращения: 21.04.2017).

[4] Смирнов Д. С. Статистический анализ эффективности контекстной рекламы // Процессы управления и устойчивость. 2015. Т. 2. № 1. С. 714–719.

[5] Смирнов Д. С. Тестирование интернет-страниц как решение задачи о многоруком бандите // Молодой ученый. 2015. № 19. С. 78–86.

[6] Смирнов Д. С. Использование задачи о многоруком бандите в тестировании веб-страниц // Процессы управления и устойчивость. 2016. Т. 3. № 1. С. 705–710.

[7] Abe N., Biermann A. W., Long P. M. Reinforcement learning with immediate rewards and linear hypotheses // Algorithmica. 2003. Т. 37. №. 4. С. 263–293.

[8] Agrawal S., Goyal N. Analysis of thompson sampling for the multi-armed bandit problem // Journal of Machine Learning Research: Workshop and Conference Proceedings. 2012. Vol. 23, № 39. P. 1–26.

[9] Agrawal S., Goyal N. Thompson Sampling for Contextual Bandits with Linear Payoffs // ICML (3). 2013. C. 127–135.

[10] Auer P., Cesa-Bianchi N., Fischer P. Finite-time Analysis of the Multiarmed Bandit Problem // Machine Learning. 2002. Vol. 47, No 2-3. P. 235–256.

[11] Auer P. et al. The nonstochastic multiarmed bandit problem // SIAM journal on computing. 2002. T. 32. №. 1. C. 48–77.

[12] Auer P. Using confidence bounds for exploitation-exploration trade-offs // Journal of Machine Learning Research. 2002. T. 3. №. Nov. C. 397–422.

[13] Awerbuch B., Kleinberg R. Online linear optimization and adaptive routing // Journal of Computer and System Sciences. 2008. T. 74. № 1. C. 97–114.

[14] Cesa-Bianchi N., Fischer P. Finite-time regret bounds for the multiarmed bandit problem // ICML. 1998. C. 100–108.

[15] Chu W. et al. Contextual Bandits with Linear Payoff Functions // AISTATS. 2011. T. 15. C. 208–214.

[16] Hardwick J. et al. Bandit strategies for ethical sequential allocation // Computing Science and Statistics. 1991. T. 23. № 6.1. C. 421–424.

[17] Kuleshov V., Precup D. Algorithms for the multi-armed bandit problem // Journal of Machine Learning Research. 2000. P. 1–48.

[18] Lage R. et al. Choosing which message to publish on social networks: A contextual bandit approach // Advances in Social Networks Analysis and Mining (ASONAM), 2013 IEEE/ACM International Conference on. IEEE, 2013. C. 620–627.

[19] Lai T. L., Robbins H. Asymptotically efficient adaptive allocation rules // Advances in applied mathematics. 1985. № 6. P. 4–22.

[20] Langford J., Zhang T. The epoch-greedy algorithm for multi-armed bandits with side information // Advances in neural information processing systems. 2008. C. 817–824.

[21] Li L. et al. A contextual-bandit approach to personalized news article recommendation // Proceedings of the 19th international conference on World wide web. ACM, 2010. C. 661–670.

[22] Lu T., Pal D., Pal M. Contextual Multi-Armed Bandits // AISTATS. 2010. C. 485–492.

[23] Pandey S., Olston C. Handling advertisements of unknown quality in search advertising // NIPS. 2006. T. 20. C. 1065–1072.

[24] Robbins H. Some aspects of the sequential design of experiments // Herbert Robbins Selected Papers. Springer New York, 1985. C. 169–177.

[25] Schwartz E. M., Misra K., Abernethy J. Dynamic Online Pricing with Incomplete Information Using Multi-Armed Bandit Experiments. 2016.

[26] Scott S. L. A modern Bayesian look at the multi-armed bandit // Applied Stochastic Models in Business and Industry. 2010. Vol. 26, № 6. P. 639–658.

[27] Shen W. et al. Portfolio Choices with Orthogonal Bandit Learning // IJCAI. 2015. C. 974–980.

[28] Strehl A. L. et al. Experience-efficient learning in associative bandit problems // Proceedings of the 23rd international conference on Machine learning. ACM, 2006. C. 889–896.

[29] Sutton R. S., Barto A. G. Reinforcement learning: An introduction. Cambridge : MIT press, 1998. T. 1. №. 1.

[30] Thompson W. R. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples // Biometrika. 1933. T. 25. №. 3/4. C. 285–294.

[31] Walsh T. J. et al. Exploring compact reinforcement-learning representations with linear regression // Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence. AUAI Press, 2009. C. 591–598.

[32] Woodroofe M. A one-armed bandit problem with a concomitant variable // Journal of the American Statistical Association. 1979. T. 74. №. 368. C. 799–806.

6. Приложения

6.1. Приложение 1. Программный код (глава 1)

```
def show_page(number_page, reward_show, number_show):  # Show page
  reward_show[number_page] += bernoulli.rvs(p[number_page])
  number_show[number_page] += 1

def regret(par_list):  # Calc regret
  temp_best_index_page = p.index(max(p))
  temp_regret_list = [0 for i in range(n_show)]
  for i in range(n_show):
    j, temp_var = 0, 0
    while j < i:
      temp_var += p[par_list[j]]
      j += 1
    temp_regret_list[i] = i * p[temp_best_index_page] - temp_var
  return temp_regret_list

def e_greedy(epsilon):  # Epsilon-greedy
  x = random.random()
  for i in range(n):
    ctr_e_greedy[i] = reward_show_e_greedy[i] / number_show_e_greedy[i]
  best_page_index = ctr_e_greedy.index(max(ctr_e_greedy))
  if x < 1 - epsilon:
    return best_page_index
  else:
    best_page_index = random.randint(0, n - 1)
    return best_page_index

def en_greedy(j):  # Epsilon-n greedy
  x = random.random()
  par_epsilon = min(c * n / (delta * delta * j), 1)
  for i in range(n):
    ctr_en_greedy[i] = reward_show_en_greedy[i] / number_show_en_greedy[i]
  best_page_index = ctr_en_greedy.index(max(ctr_en_greedy))
  if x < 1 - par_epsilon:
```

```
    return best_page_index
  else:
    best_page_index = random.randint(0, n - 1)
    return best_page_index

def ucb(t):  # UCB1
  for i in range(n):
    ctr_ucb[i] = reward_show_ucb[i] / number_show_ucb[i]
  max_temp = reward_show_ucb[0] / number_show_ucb[0] + math.sqrt(2 * math.log(t)
     / number_show_ucb[0])
  best_page_index = 0
  temp = [0 for i in range(n)]
  for i in range(n):
    temp[i] = reward_show_ucb[i] / number_show_ucb[i] + math.sqrt(2 * math.log(t)
        / number_show_ucb[i])
    if temp[i] > max_temp:
      max_temp = temp[i]
      best_page_index = i
  return best_page_index

def thompson_sampling():  # Thompson Sampling
  for i in range(n):
    ctr_sampling[i] = reward_show_sampling[i] / number_show_sampling[i]
  maximum, best_page_index, i = 0, 0, 0
  while i < n:
    beta_rasp = random.betavariate(reward_show_sampling[i] + 1,
       number_show_sampling[i] - reward_show_sampling[i] + 1)
    if beta_rasp > maximum:
      maximum, best_page_index = beta_rasp, i
    i += 1
  return best_page_index

def softmax(tau):  # Softmax
  for i in range(n):
    ctr_softmax[i] = reward_show_softmax[i] / number_show_softmax[i]
  p1 = [0 for i in range(n)]
  summ, j, best_page_index = 0, 0, 0
  while j < n:
```

```
        summ += math.exp(reward_show_softmax[j] / number_show_softmax[j] / tau)
        j += 1
    i = 0
    while i < n:
        p1[i] = math.exp(reward_show_softmax[i] / number_show_softmax[i] / tau) /
            summ
        i += 1
    x = random.random()
    temp_sum, i, best_page_index  = p1[0], 1, 0
    while i <= n:
        if x < temp_sum:
            best_page_index = i - 1
            break
        else:
            temp_sum += p1[i]
            i += 1
    return best_page_index

def pursuit(beta, t):  # Pursuit
    for i in range(n):
        ctr_pursuit[i] = reward_show_pursuit[i] / number_show_pursuit[i]
    best_page_index = ctr_pursuit.index(max(ctr_pursuit))
    if t == 1:
        for i in range(n):
            pursuit_list[i] = 1 / n
    else:
        j = 0
        while j < n:
            if j == best_page_index:
                pursuit_list[j] += beta * (1 - pursuit_list[j])
            else:
                pursuit_list[j] += beta * (0 - pursuit_list[j])
            j += 1
    x = random.random()
    temp_sum, i, best_page_index  = pursuit_list[0], 1, 0
    while i <= n:
        if x < temp_sum:
            best_page_index = i - 1
```

```
      break
    else:
      temp_sum += pursuit_list[i]
      i += 1
  return best_page_index

def play_winner(t, flag):  # Play the winner
  for i in range(n):
    ctr_winner[i] = reward_show_winner[i] / number_show_winner[i]
  if flag:
    best_page_index = page_in_time_winner[t - 1]
  else:
    best_page_index = random.randint(0, n - 1)
  return best_page_index

def comparison(t):  # Reinforcement Comparison
  for i in range(n):
    ctr_comparison[i] = reward_show_comparison[i] / number_show_comparison[i]
  p1 = [0 for i in range(n)]
  summ, j = 0, 0
  while j < n:
    summ += math.exp(comparison_pi[j][t])
    j += 1
  i = 0
  while i < n:
    p1[i] = math.exp(comparison_pi[i][t]) / summ
    i += 1
  x = random.random()
  temp_sum, i, best_page_index  = p1[0], 1, 0
  while i <= n:
    if x < temp_sum:
      best_page_index = i - 1
      break
    else:
      temp_sum += p1[i]
      i += 1
  return best_page_index
```

6.2. Приложение 2. Программный код (глава 2)

```
import math
import numpy as np
import numpy.linalg as lal
import matplotlib.pyplot as plt
import random
from scipy.stats import bernoulli

p_context = [0.2, 0.2, 0.2, 0.2]
p = []
m = 0  # num of ads

# read data
def read(f):
  global m
  m = 0
  for line in f:
    m += 1
    row = [float(i) for i in line.split()]
    p.append(row)
  return 0

f = open('contextual_in.txt', 'r')
read(f)
n = len(p[0]) # num of pages
t = 1000  # num of shows
launch_number = 50  # num of runs
alpha = 3

# generate context
def context():
  x = random.random()
  temp_sum = p_context[0]
  i = 1
  number_of_ad = 0
  while i <= n:
    if x < temp_sum:
```

```
      number_of_ad = i - 1
      break
    else:
      temp_sum += p_context[i]
      i += 1
  return number_of_ad
  number = random.randint(0, m - 1)
  return number

# func of show page
def show_page(index_page):
  current_reward = bernoulli.rvs(p_main[index_page])
  reward_show_main[index_page] += current_reward
  number_show_main[index_page] += 1
  reward_show[ad_number][index_page] += current_reward
  number_show[ad_number][index_page] += 1
  return current_reward

# calc regret
def func_regret_calculation(par_list1, par_list2):
  for i in range(t):
    j, temp_var_best, temp_var_current = 0, 0, 0
    while j < i:
      temp_var_best += par_list1[j]
      temp_var_current += par_list2[j]
      j += 1
    regret_list[i] = temp_var_best - temp_var_current
  return 0

a = [0 for i in range(n)]
b = [0 for i in range(n)]
teta = [0 for i in range(n)]
sum_conversion_page = [[0 for i in range(n)] for i in range(m)]
sum_regret_list = [0 for i in range(t)]
sum_mean_conversion_list = [0 for i in range(t)]
p_main = [0 for i in range(n)]
regret_list = [0 for i in range(t)]
x = [[0 for i in range(1)] for j in range(m)]  # context vector
```

```
p_alg = [0 for i in range(n)]
sum_number_context = [0 for i in range(m)]
sum_number_show = [[0 for i in range(n)] for i in range(m)]

# main loop
launch_count = 0
while launch_count < launch_number:
  number_show = [[1 for i in range(n)] for i in range(m)]
  reward_show = [[0 for i in range(n)] for i in range(m)]
  reward_show_main = [0 for i in range(n)]
  number_show_main = [0 for i in range(n)]
  conversion_page = [[0 for i in range(n)] for i in range(m)]
  best_conversion = [0 for i in range(t)]
  current_conversion = [0 for i in range(t)]
  regret_list = [0 for i in range(t)]
  mean_conversion_list = [0 for i in range(t)]
  number_context = [0 for i in range(m)]

  # matrix
  for i in range(n):
    a[i] = np.eye(m)
    b[i] = np.matrix([[0 for j in range(1)] for k in range(m)])
    teta[i] = np.matrix([[0 for j in range(1)] for k in range(m)])

  time = 1
  # inner loop
  while time < t:
    ad_number = context()
    number_context[ad_number] += 1
    x = np.matrix([[0 for i in range(1)] for j in range(m)])
    x[ad_number][0] = 1
    for i in range(n):
      p_main[i] = p[ad_number][i]
    max_p = 0
    max_i = 0
    mean_conversion = 0
    best_conversion[time] = max(p_main)
    for i in range(n):
```

```
      teta[i] = lal.linalg.inv(a[i]) * b[i]
      p_alg[i] = teta[i].transpose() * x + alpha * math.sqrt(x.transpose() * lal.
          linalg.inv(a[i]) * x)
      if p_alg[i] > max_p:
        max_p = p_alg[i]
        max_i = i
    current_conversion[time] = p_main[max_i]
    reward = show_page(max_i)
    a[max_i] += x * x.transpose()
    b[max_i] += reward * x
    for i in range(n):
      mean_conversion += reward_show_main[i]
    mean_conversion_list[time] = mean_conversion / time

    time += 1

  func_regret_calculation(best_conversion, current_conversion)

  for k in range(t):
    sum_regret_list[k] += regret_list[k]
    sum_mean_conversion_list[k] += mean_conversion_list[k]

  for i in range(m):
    sum_number_context[i] += number_context[i]
    for j in range(n):
      conversion_page[i][j] = reward_show[i][j] / number_show[i][j]
      sum_conversion_page[i][j] += conversion_page[i][j]
      sum_number_show[i][j] += number_show[i][j]

  launch_count += 1


for i in range(t):
  sum_regret_list[i] /= launch_number
  sum_mean_conversion_list[i] /= launch_number

for i in range(m):
  sum_number_context[i] /= launch_number
```

```
  for j in range(n):
    sum_conversion_page[i][j] /= launch_number
    sum_number_show[i][j] /= launch_number

# the best page
mean_of_conversion = [0 for i in range(n)]
for i in range(n):
  suma = 0
  for j in range(m):
    suma += sum_number_context[j] * sum_conversion_page[j][i]
  mean_of_conversion[i] = suma

print('all conv = ', mean_of_conversion)
print('best page is ', mean_of_conversion.index(max(mean_of_conversion)))
print('conversion of page = ', sum_conversion_page)
print('number of show = ', sum_number_show)
print('number of context = ', sum_number_context)

time_list = [i for i in range(t)]
plt.figure(1)
plt.plot(time_list, sum_regret_list, linestyle='--', label='LinUCB')
plt.title('LinUCB', fontsize=18)
plt.xlabel('time', fontsize=16)
plt.ylabel('regret', fontsize=16)
plt.legend(loc='upper left', prop={'size': 14})
plt.grid(True)
plt.show()

plt.figure(2)
plt.plot(time_list, sum_mean_conversion_list, linestyle='--', color='blue', label
   ='LinUCB')
plt.title('LinUCB', fontsize=18)
plt.xlabel('time', fontsize=16)
plt.ylabel('conversion', fontsize=16)
plt.legend(loc='upper left', prop={'size': 14})
plt.grid(True)
plt.show()
```

6.3. Приложение 3. Программный код (глава 3)

```
import math
import matplotlib.pyplot as plt
from scipy.stats import bernoulli

# actions, experts
p1, p2 = [0.3, 0.45, 0.7, 0.47, 0.1], [0.3, 0.5, 0.1, 0.5, 0.1]
n = len(p1)
launch_number, time_finish = 10, 1000
index_best_arm = p1.index(max(p1))

def func_win():
  mean_win[current_number] += bernoulli.rvs(p1[current_number])
  num_of_games[current_number] += 1

def func_first_win():
  for i in range(n):
    mean_win[i] = bernoulli.rvs(p1[i])

def number_of_arms(t):
  global current_number
  max_temp, max_temp_index = mean_win[0] / num_of_games[0] + math.sqrt(2 * math.
      log(t) / num_of_games[0]), 0
  for i in range(n):
    temp[i] = mean_win[i] / num_of_games[i] + math.sqrt(2 * math.log(t) /
        num_of_games[i])
    if temp[i] > max_temp:
      max_temp, max_temp_index = temp[i], i
  current_number = max_temp_index

def func_regret_calculation():
  for i in range(time_finish):
    j, temp_var = 0, 0
    while j < i:
      temp_var += p1[current_number_vector[j]]
      j += 1
    regret_vec[i] = i * p1[index_best_arm] - temp_var
```

```
def func_forecast():
  for i in range(n):
    current_forecast_vec[i] = bernoulli.rvs(p2[i])
    forecast_vec[i] += current_forecast_vec[i]

def number_of_arms_f(t):
  global current_number
  max_temp, max_temp_index = mean_win[0] / num_of_games[0] + math.sqrt(2 * math.
     log(t) / num_of_games[0]) + forecast_vec[0] / t * math.exp(-abs(mean_win[0]
      / num_of_games[0] - forecast_vec[0] / t)), 0
  for i in range(n):
    temp[i] = mean_win[i] / num_of_games[i] + math.sqrt(2 * math.log(t) /
       num_of_games[i]) + forecast_vec[i] / t * math.exp(-abs(mean_win[i] /
       num_of_games[i] - forecast_vec[i] / t))
    if temp[i] > max_temp:
      max_temp, max_temp_index = temp[i], i
  current_number = max_temp_index

sum_of_reward = [0 for i in range(n)]
mean_regret_vec = [0 for i in range(time_finish)]
regret_vec = [0 for i in range(time_finish)]
mean_win = [0 for i in range(n)]
num_of_games = [1 for i in range(n)]
current_number_vector = [0 for i in range(time_finish)]
temp = [0 for i in range(n)]

launch_count = 0
while launch_count < launch_number:
  count, current_number = 1, 0
  for i in range(n):
    mean_win[i], num_of_games[i], temp[i] = 0, 1, 0
  for i in range(time_finish):
    regret_vec[i], current_number_vector[i] = 0, 0

  func_first_win()

  while count < time_finish:
```

```
    number_of_arms(count)
    current_number_vector[count] = current_number
    func_win()
    count += 1

  func_regret_calculation()

  for i in range(time_finish):
    mean_regret_vec[i] += regret_vec[i]
  for i in range(n):
    sum_of_reward[i] += mean_win[i]

  launch_count += 1

sum_reward = 0
for i in range(n):
  sum_reward += sum_of_reward[i] / launch_number
print('Reward UCB1 = ', sum_reward)

for i in range(time_finish):
  mean_regret_vec[i] /= launch_number

mean_regret_vec_classic = [0 for i in range(time_finish)]
for i in range(time_finish):
  mean_regret_vec_classic[i] = mean_regret_vec[i]
  mean_regret_vec[i] = 0

sum_of_reward_f = [0 for i in range(n)]
forecast_vec = [0 for i in range(n)]
current_forecast_vec = [0 for i in range(n)]

launch_count = 0
while launch_count < launch_number:
  count = 1
  for i in range(n):
    mean_win[i], num_of_games[i], forecast_vec[i], current_forecast_vec[i] = 0,
        1, 0, 0
  for i in range(time_finish):
```

```
    regret_vec[i], current_number_vector[i] = 0, 0
  func_first_win()
  while count < time_finish:
    func_forecast()
    number_of_arms_f(count)
    func_win()
    current_number_vector[count] = current_number
    count += 1

  for i in range(n):
    sum_of_reward_f[i] += mean_win[i]

  func_regret_calculation()
  for i in range(time_finish):
    mean_regret_vec[i] += regret_vec[i]

  launch_count += 1

sum_reward = 0
for i in range(n):
  sum_reward += sum_of_reward_f[i] / launch_number
print('Reward with an expert = ', sum_reward)

for i in range(time_finish):
  mean_regret_vec[i] /= launch_number

time = [i for i in range(time_finish)]
plt.figure(1)
plt.plot(time, mean_regret_vec_classic,linestyle='--',label='UCB1='+str(p1))
plt.plot(time, mean_regret_vec, linestyle='-',color='red',label='e1='+str(p2))
plt.title('The total expected regret (UCB1 Bernoulli)', fontsize=18)
plt.xlabel('time', fontsize=16)
plt.ylabel('regret', fontsize=16)
plt.legend(loc='upper left', prop={'size': 12})
plt.grid(True)
plt.show()
```

6.4. Приложение 4. Программный код (глава 3)

```
#method 1
def calc_vec_mera(t):
  ro = [0 for i in range(m)]
  for i in range(m):
    kvad_razn = 0
    for j in range(n):
      kvad_razn += (forecast_matr[i][j] / t - mean_win[j] / num_of_games[j]) * (
          forecast_matr[i][j] / t - mean_win[j] / num_of_games[j])
    ro[i] = math.sqrt(kvad_razn)
  temp_min, temp_index_min = ro[0], 0
  for i in range(m):
    if ro[i] < temp_min:
      temp_min = ro[i]
      temp_index_min = i
  for j in range(n):
    forecast_vec[j] = forecast_matr[temp_index_min][j]
  num_of_games_expert[temp_index_min] += 1
  return 0

# method 2
def calc_vec_best_accr(t):
  accur_matr = [[0 for i in range(n)] for j in range(m)]
  for i in range(m):
    for j in range(n):
      accur_matr[i][j] = abs(forecast_matr[i][j] / t - mean_win[j] / num_of_games
          [j])
  for i in range(n):
    temp_min, temp_index_min = accur_matr[0][i], 0
    for j in range(m):
      if accur_matr[j][i] < temp_min:
        temp_min = accur_matr[j][i]
        temp_index_min = j
    forecast_vec[i] = forecast_matr[temp_index_min][i]
    num_of_choice_accur[temp_index_min][i] += 1
  return 0
```

```
# method 3
def choice_expert(t):
  global current_number_expert
  max_temp_expert = mean_win_expert[0] / num_of_games_expert[0] + math.sqrt(2 *
     math.log(t) / num_of_games_expert[0])
  max_temp_index_expert = 0
  for k in range(m):
    temp_expert[k] = mean_win_expert[k] / num_of_games_expert[k] + math.sqrt(2 *
       math.log(t) / num_of_games_expert[k])
    if temp_expert[k] > max_temp_expert:
      max_temp_expert = temp_expert[k]
      max_temp_index_expert = k
  current_number_expert = max_temp_index_expert
  num_of_games_expert[current_number_expert] += 1
  return 0
```